NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEVELOPMENT AND ANALYSIS OF SECURITY POLICIES IN SECURITY ENHANCED ANDROID

by

Ryan A. Rimando

December 2012

Thesis Advisor:	George W. Dinolt
Second Reader:	Karen Burke

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 2012	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Development and Analysis of Security Policies in Security Enhanced Android	5. FUNDING NUMBERS
6. AUTHOR(S) Rimando, Ryan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943–5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT (maximum 200 words)

This thesis examines Security Enhanced Android. Both its policy and its additional security features are explored. The policy is examined in depth, providing a better understanding of the security provided by SE Android. We analyze the default SE Android policy. We identify a potential weakness and change the policy to facilitate control over communication channels. A proof-of-concept set of applications is developed to demonstrate how SE Android can be used to improve application security. The proof-of-concept policy is then analyzed to determine if security goals are met.

14. SUBJECT TERMS Android, SE Android, SE Linux, Security Policy	15. NUMBER OF PAGES 121
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540–01–280–5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DEVELOPMENT AND ANALYSIS OF SECURITY POLICIES IN SECURITY ENHANCED ANDROID

Ryan A. Rimando
Civilian, Federal Cyber Corps
B.S., College of Charleston, 2010

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2012

Author: Ryan A. Rimando

Approved by: George W. Dinolt
Thesis Advisor

Karen Burke
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis examines Security Enhanced Android. Both its policy and its additional security features are explored. The policy is examined in depth, providing a better understanding of the security provided by SE Android. We analyze the default SE Android policy. We identify a potential weakness and change the policy to facilitate control over communication channels. A proof-of-concept set of applications is developed to demonstrate how SE Android can be used to improve application security. The proof-of-concept policy is then analyzed to determine if security goals are met.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ACL	Access Control List
AOSP	Android Open Source Project
AVR	Access Vector Rule
DAC	Discretionary Access Control
DDT	Domain Definition Table
DIT	Domain Interaction Table
Flask	Flux Advanced Security Kernel
IDE	Integrated Development Environment
IPC	Inter-process Communication
MAC	Mandatory Access Control
MCS	Multi-Category Security
MLS	Multi-Level Security
PID	Process Id
POLP	Principle of least privilege
RBAC	Role-based Access Control
SE	Security Enhanced
TE	Type Enforcement
UID	User Id

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. George Dinolt, and my second reader, Karen Burke, for their help and support throughout the thesis process. I am grateful to the National Science Foundation's Scholarship for Service (SFS) Program and the opportunities it has provided me. I would also like to thank Dr. Cynthia Irvine, the Naval Postgraduate School SFS Principal Investigator, and Valerie Linhoff for all their work in the SFS program that made my time at Naval Postgraduate School an enjoyable experience.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. DISCUSSION

Mobile computing is becoming increasingly more prevalent in the world today. From smartphones to tablets, people are using mobile computing more and more. Now, nearly half of adults in the U.S. own smartphones [1]. This also provides attackers with a new vector of attack for exploiting devices and obtaining user information. Thus, there is a need for better mobile security.

There are a number of platforms from BlackBerrys to iPhones for attackers to target. Attackers tend to pick the easiest and largest target possible. With Android's U.S. market share being around 50% [2] and it's completely open source nature, it is a prime target for attackers. Trend Micro reports that the number of malicious applications in Q2 of 2012 was around 20,000 [3]. They also note that these malicious applications were downloaded 700,000 times before Google was able to remove many them. Researchers have been and are working on a number of techniques to counter this threat. Recently the NSA (U.S. National Security Agency) became involved in this process.

Early in 2012, the NSA made their first public release of Security Enhanced Android, or SE Android. SE Android's aim is to improve Android security through the introduction of MAC and other security enhancements.

B. SCOPE

This thesis contributes to the currently limited literature on SE Android. With SE Android still being in its infancy, there is much more work that can and should be done. The scope of this thesis is a study of SE Android and how to use its added security functionality to improve application security. This research examines SE Android's security policy and how it interacts with Android. A proof-of-concept application is developed to demonstrate how the policy can be adjusted to provide security for specific applications. In the process of developing the proof-of-concept we analyze the original

SE Android security policy and modify it to facilitate control of communication channels. We identify a weakness in the current SE Android security policy and provide changes to enhance security.

C. ORGANIZATION OF THESIS

General background on information security is presented in Chapter II. Also included in that chapter are brief histories of SE Linux, Android and SE Android for those unfamiliar with them. Chapter III covers the architecture of Android as well as its various security mechanisms. Chapter IV briefly covers SE Linux's type enforcement and role-based access control policies and implementations. Chapter V provides an in-depth look at SE Android and how it works to improve security in Android. Chapter VI covers a proof-of-concept application, which utilizes the enhanced security provided by SE Android. Chapter VII is contains a summary and suggestions for future work.

II. BACKGROUND

A. INTRODUCTION

In this chapter we provide some background that will be useful in later parts of the thesis. We begin by introducing some basic security principles. This will be followed by a brief summary of Access Control followed by descriptions of three specific models relevant to this thesis. Then, brief histories of SE Linux, Android, and SE Android will be covered. Lastly, a summary of related works will be given.

B. CIA PRINCIPLES

Information security is the protection of information and information systems from unauthorized usage. It is based around three primary principles. These principles are often referred to as the CIA triad:

- **Confidentiality** – refers to the protection of information from access by unauthorized individuals
- **Integrity** – involves the protection of information against unauthorized modification or deletion
- **Availability** – the assurance that information is accessible in a timely fashion to authorized individuals

These three principles are at the core of information security. An argument can be made for adding to these principles. Two common proposals are authenticity and non-repudiation. Authenticity is the validation that a communicating party is who they claim they are. Non-repudiation is a guarantee that the sender of a message cannot deny having sent the message and similarly the recipient cannot deny receiving it.

The security and privacy of information while stored, processed, or transmitted is protected by preserving these principles in information systems. One of the mechanisms used to preserve these principles is access control.

C. ACCESS CONTROL

Access control is the mechanism by which the system provides both Confidentiality and some forms of Integrity security. Access control is perhaps the most

fundamental security mechanism used today. In general, access control models consist of three basic elements:

- **Subjects** – the actors within a system. Primarily they are the processes or other elements that do "processing."
- **Objects** – the items on which subjects operate. Typically these are files or file like objects that store information.
- **Actions** – the operations subjects perform on objects. These actions can include things like reading, writing, and executing.

It is the relationship of these elements that serves as the basis for access control models. These relationships are usually represented in Access Control Lists (ACLs). ACLs contain the allowed permissions for each subject of a system. Every subject is allowed to perform only the specified actions on any given object as specified in the ACL. In systems where ACLs are implemented, every time a subject wishes to perform an action on an object, the system must check the ACL. The mechanism that performs these checks is essential to ensuring the security provided by access control. That mechanism is called the reference monitor.

The reference monitor was first introduced in the "Anderson Report" in 1972 [4]. This influential computer security paper defined three principles that reference monitors need to follow:

- **Tamper-proof** – The reference monitor must be protected from interference or tampering by an attacker. This ensures that the reference monitor properly enforces the security policy
- **Completeness** – The reference monitor must always be invoked. Every operation subjects perform on objects must be validated by the reference monitor lest an attacker bypass and therefore violate the security policy
- **Verifiable** – The reference monitor must be small and simple enough to be properly analyzed and tested. This ensures that the mechanism is not flawed and properly enforces the security policy

Many different access control models have been developed. There are two main approaches on how access control is implemented: Discretionary Access Control and Mandatory Access Control. We discuss those briefly along with Capability-based systems.

1. Discretionary Access Control

Discretionary Access Control, or DAC, is currently the most prevalent form of access control. In DAC, access is granted at the discretion of the owner or creator of an object. In other words, if John creates a file, he can choose whether or not to give Jane access and what kind of access she has.

DAC mechanisms are generally recognized as an inadequate protection mechanism when used as the sole method of access control. The discretion at which permissions can be passed between users makes the potential for vulnerabilities much higher than in MAC models. For instance, if an application that is running on behalf of a user is compromised by an attacker, then the attacker will be able to modify all the resources that user has access to. This is why applications running as "root" on Unix like systems or "Administrator" on Windows based systems are so dangerous in a DAC environment. Nevertheless, the ease of implementing and the granularity of control available make DAC an attractive access control model.

2. Mandatory Access Control

Mandatory Access Control, or MAC, allows for more controlled access control. In MAC, access is granted based on some defined security policy enforced by the underlying system. Access control is not left to the discretion of the originator. Users must adhere to the rules set forth by the security policy. There are a number of different models for MAC based access control: Bell-LaPadula, Biba, Clark-Wilson, Role-Based Access Control, Type Enforcement, etc. For the purposes of this thesis, we will discuss the Role-Based Access Control models and the Type Enforcement model as these are the models at work in SE Linux. Additionally, the Bell-LaPadula model will be discussed as it is found in the MLS extension for both SE Linux and SE Android.

a. Role-Based Access Control Model

Role-Based Access Control (RBAC) was originally introduced in 1992 by Ferraiolo and Kuhn [5]. RBAC provides access to objects based on a user's role. This idea of roles is analogous to roles in an organization. For instance, accountants and

salesman would be separate roles. By grouping users based on their roles, the size and complexity of security policy implementation is vastly reduced. Instead of assigning permissions to individual users, permissions are assigned to roles, which users/subjects may be a member. The major elements in RBAC are:

- **Users** – human or process
- **Role** – job function within context of organization or system
- **Permission** – operation rights on objects
- **Session** – instance of a user's connection to the system

Figure 1 shows the relationships between these elements. Users can be assigned multiple roles. Similarly, Roles can contain any number of users. Roles are assigned a number of permissions.

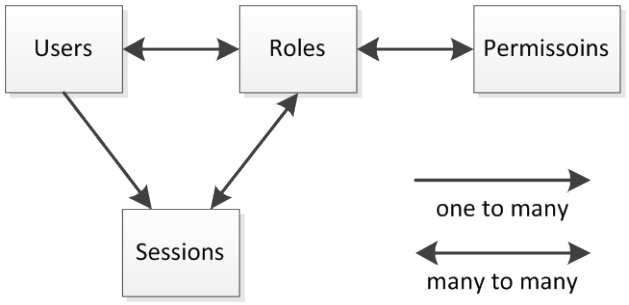

Figure 1. RBAC element relationships

In RBAC, operations performed by users are called transactions. There are three basic rules in RBAC as set forth in [6]:

- **Role assignment** – Subjects can only execute transactions if that subject has selected or been assigned a role. Identification and authentication is not considered a transaction, but all other operations are. Therefore, all active users must have some active role.
- **Role authorization** – The active role of a subject must have been authorized for that subject. Thus, users can only take on roles for which they have been authorized.

- **Transaction authorization** – For a subject to execute a transaction, that transaction must be authorized for the subject's active role. This, combined with role assignment and authorization, ensures that users can only execute those transactions for which they are authorized.

There are several different models or levels of RBAC that were introduced in a framework called RBAC96 [7]. $RBAC_0$ is the basic model of RBAC. It embodies the basic principles of RBAC: users are assigned to roles, permissions are assigned to roles, and thus users acquire the permissions of their given roles.

$RBAC_1$ is $RBAC_0$ with role hierarchies. The role hierarchy allows roles to subsume others. In other words, a user assigned to a role R has access to any sub-roles of R.

$RBAC_2$ enhances $RBAC_0$ by introducing constraints. Constraints are used in a number of different ways. Mutual exclusivity constraints can restrict a user from being a member of two roles, restrict two roles from having the same permissions, restrict one role from have two permissions. Cardinality constraints can place limits on how many users are a member of a role. Prerequisite constraints can require role membership of a role A only if the user is a member of role B. These constraints can be used to ensure separation of duties. $RBAC_3$ is simply a combination of the features provided by $RBAC_1$ and $RBAC_2$.

b. Type Enforcement Model

Type Enforcement (TE) is a very flexible and fine-grained security model [54]. Like most other models, TE divides system elements into subjects and objects. In TE, every subject and object is labeled. Subjects are given domain labels and objects have type labels. These labels are referred to as security contexts. Access control is enforced by comparing a subject's security context against an objects security context. Access can be granted between two domains or from a domain to a type; so essentially there are two groups of permission types: domain-domain and domain-type.

The TE model permission map is represented in two tables: the Domain Definition Table (DDT) for domain-type relations and the Domain Interaction Tables

(DIT) for domain-domain relations. Table 1 represents an example of a Domain Definition Table. Subjects, the rows, can access objects, the columns, based on the permissions indicated in the cells. For instance, the ftp process domain has read and write access to public files.

	File Types			
Process Domains	Web pages	Mailbox file	Mail Aliases	Public files
WWW Server	r			
Mail System		rw	r	
ftp				rw

r=read; w=write; blank squares indicate no access is allowed

Table 1. Domain Definition Table, from [8]

One of the goals of the TE model is to control the flow of information from one process to another. By controlling access to the objects or containers, the flow of information can be tracked by observing how the subjects are allowed to interact with the objects. For instance, consider subjects A and B and object C. If subject A has write access to object C and B has read access to object C, then subject A can pass information to subject B through object C.

c. Bell-LaPadula Model

The Bell-LaPadula model is probably the most well-known model for MAC. Introduced in 1973, it focuses on protecting the confidentiality of classified information [9]. As with most all other models, the Bell-LaPadula model splits elements in the system into subjects and objects. Subjects are the active elements in the system that can act upon the passive elements, objects.

Each subject and object is labeled with a security attribute. A simplified version of this model labels subjects and objects with two elements: a sensitivity level, and a category. For subjects, the sensitivity level can be thought of as a security clearance, and similarly for objects, the sensitivity level corresponds to a security classification. In many implementations, sensitivity levels correspond to *Confidential*,

Secret, and *Top Secret*. Sensitivity levels fall in a hierarchy with *Secret* dominating *Confidential* and *Top Secret* dominating *Secret* and by transitivity *Confidential*.

Categories, or compartments, serve to further isolate information following the need-to-know principle. Within each sensitivity level, there may be any number of categories. For instance, at the *Secret* level there may be data for troop locations and weather data. We may not want everyone at the *Secret* level to be able to read the troop location data and using categories allows for that.

These labels and compartments are organized in a partially ordered set with a least upper and greatest lower bound called a lattice [29]. In a lattice, higher sensitivity levels dominate lower levels (e.g. *Secret* dominates *Confidential*).

There are three key properties to the Bell-LaPadula model:

- **Simple Security Property** – This property specifies that a subject cannot read any objects that are at a higher sensitivity level. This is commonly referred to as no read-up.
- **Star Property –** This property states that a subject cannot write to any objects that are at a lower sensitivity level than itself. This is commonly referred to as no write-down.
- **Discretionary Security Property –** This states that an access matrix is used to specify additional access controls. Note that this property does not mean that access is at the discretion of the owners.

When all of these properties are satisfied, the confidentiality of the information is ensured. A subject at the *Secret* level cannot read data from the *Top Secret* level nor can it write information down to the C*onfidential* level.

3. Capability-based Systems

Capability-based security is a security model that relies on, as the name suggests, capabilities. The idea behind it is fairly simple. Security comes from the handling of capabilities, which are commonly described as analogous to keys. While that analogy is not entirely accurate [27], it serves our purposes for comparing it to binder – a service used to manage access in Android and SE Android, which we describe in detail below. Capabilities are values that reference an object along with its associated rights, similar to

a key being associated with a particular lock. In capability systems, these keys are un-forgeable. Thus the only way to obtain the key is to either be the original creator, or receive it from some entity that already has a copy of the key.

D. SECURITY ENHANCED LINUX

SE Linux was officially released by the NSA in 2000. Based on the Flask Security Architecture [10], SE Linux is designed to provide a flexible MAC solution capable of supporting a variety of security policies. It has since been adapted to use the Linux Security Module framework, an extension of the Linux kernel, and was officially adopted in the mainline kernel in 2003. Now, SE Linux is included in a number of Linux distributions including Fedora, Red Hat, and Ubuntu [55–57] and actively used in other systems such as EnGarde Secure Linux and Chromium OS [64, 65].

SE Linux makes use of MAC, TE and RBAC models. All subjects and objects in the system are assigned security labels, which will be used to determine access rights. RBAC is layered on top of TE to use roles in the grouping domain types. The specifics of SE Linux implementation of these models are discussed in Chapter IV.

The core feature of SE Linux that makes it so widely used is its flexible policy configuration. SE Linux policies consist of any number of configuration files that contain all the rules to be enforced by the kernel. Unfortunately, the level of granularity SE Linux provides often causes these policies to become very large making it difficult for administrators to configure and understand the policy that is implemented. A number of tools have been developed to aide in the handling of the development and analysis SE Linux policies and the configuration files used to implement them.

E. ANDROID

Android is an operating system designed for smartphones and other mobile devices. Android, Inc. was founded in 2003 to develop smarter mobile devices. Now owned by Google, Android is completely open-source and maintained under the Android Open Source Project. Android launched commercially in 2008 and has gone through a number of versions since. The latest stable version is Jelly Bean, or Android 4.1, following the typical convention of naming versions after some kind of food. Android

continues to evolve with new features, and lately it has been modified to support tablet-like features.

As of 2010, it is the leading smartphone platform in the world. Some of this success can be attributed to its rich community of application developers. The Android Market is now part of Google Play[1], Google's online store, and houses half a million applications as of May 2012. With the goal of being a developer friendly platform, Android has a relaxed vetting process for pushing applications to market, making it easy for developers to get their products to users.

Unfortunately, the ease with which applications can be introduced to the Android Market also increases the risk of malicious applications proliferating to devices. In order to minimize the impact of the inevitable malicious applications, Android provides a number of security mechanisms to mitigate the threat, including having the user involved in the determination and application of their security model. Android's security mechanisms will be discussed in further detail in Chapter III.

In addition to security mechanisms in the OS, the Android Market also has the capability of remotely removing malware from devices. Should malware be identified on the Market, this capability allows for rapid and scalable remediation without requiring the users to be involved.

Early in 2012, Google announced that they were scanning Android applications for malware using what they called, Bouncer [11]. Bouncer is an automated scanning tool that traverses the Android Market looking for malicious software. Bouncer supposedly provides some protection against malicious applications while keeping with the Android theme of being a developer friendly platform. It performs a number of analyses on new applications using both static and dynamic techniques. However, during Black Hat 2012, Percoco and Schulte demonstrated how Bouncer could be circumvented by using a JavaScript bridge to add new, malicious capabilities to an originally benign application [63].

[1] https://play.google.com/store/apps

F. SE ANDROID

SE Android was developed by the NSA and released in January 2012 [12]. SE Android is an ongoing project at NSA and is a work in progress [12]. With the increasing desire to make mobile devices usable in the U.S. government, SE Android was developed with the goal to provide increased security to Android by providing MAC. Since Android is based on the Linux kernel, SE Android naturally is inspired by and largely based on SE Linux. SE Android is intended to be transparent to developers and users, requiring minimal interaction.

SE Android was originally developed on Gingerbread v 2.03 of Android but is now compatible with Android OS version 4.1.1. It can be run on the emulator, both the arm and x86 versions, and, so far, on the following devices: Galaxy Nexus, Nexus S, and the Motorola Xoom [12].

According to Smalley [13], SE Android is not a fork of Android, a government-specific version of Android, a complete solution for all security concerns, nor has it been evaluated or approved for use. It is a set of security enhancements to Android focused on closing security gaps that have not yet been addressed. Like SE Linux, SE Android is intended for wide applicability and looks to be adopted into mainline Android.

There is very little literature on SE Android. Most of what exists is limited to a wiki page[2] that focuses on obtaining and installing SE Android and a set of slides that go along with a presentation at the Android Builders Summit of 2012. Additionally, there is a message board at http://marc.info/?l-selinux where questions and issues with SE Android are discussed. Aside from these sources, information on SE Android can be garnered from the source files.

G. RELATED WORK

There has been some work done in improving the security of Android and mobile

2 www.selinuxproject.org/page/SEAndroid (Active 16 October 2012)

devices in general. This section will briefly discuss some of this work. It is important to note that not all of this relates to providing MAC in Android, but it may still be of interest.

SE Android was not the first attempt at integrating SE Linux in Android. Shabtai, Fledel, and Elovici developed an SE Linux implementation for Android in 2010 [14]. Their approach is very similar to the one that now exists in SE Android. It seems, however, that their implementation of SE Linux is merely a subset of the features that are incorporated in SE Android. Nevertheless, they pointed out some of the difficulties in the integration of SE Linux into Android that were solved in SE Android. These difficulties include labeling support for the yaffs2 file system used by Android and modification of the zygote[3] source code to enable explicit labeling of its children.

There are other attempts at improving Android security using a variety of different methods. Most of these focus on the identification and elimination of malicious activity. One example would be trying to prevent jail-breaking or rooting of devices, which, while legal as of 2010 under Section 1201(a) (1) of the U.S. Copyright Law [59], is generally frowned upon. There is always a demand for rooting as this functionality allows power users full control over their devices. Another example for improving security would be the identification of the data that leaks out over the network.

TrustDroid was developed by researchers in Germany in 2011 [15]. TrustDroid's focus was to isolate groups of applications into separate domains. For instance, one may wish for corporate applications to be isolated from private user applications. This cannot be accomplished using standard Android features. TrustDroid makes this possible with domain and data isolation and its application of a special security policy and mechanisms for enforcing it. TrustDroid extends Android's middleware and kernel to provide MAC features. These extensions allow for applications to be assigned different trust levels. TrustDroid divides applications into three different levels: system applications, trusted third party applications, and untrusted third party applications. These levels, or "colors," allow a policy manager to determine an application's isolation rules.

[3] Zygote will be discussed further in a later section.

TaintDroid is a framework that can detect the leakage of sensitive information [16]. It utilizes a dynamic taint based analysis to detect information leakage [60]. It uses a data flow analysis technique using data sources and sinks. It taints private data (sources) and tracks that data as it propagates through the system. If that data reaches a sink, then it will alert the user.

XManDroid [17], or eXtended Monitoring on Android, attempts to solve the problem of IPC-based (Inter-Process Communications) privilege escalations by enforcing policies on IPC channels similar to TaintDroid. XManDroid dynamically analyzes applications' transitive permission usage in order to prevent application-level privilege escalation attacks at runtime. Unlike TaintDroid, XManDroid can detect the use of covert channels leaking sensitive information.

III. ANDROID AND ITS SECURITY FEATURES

This chapter is intended to provide a brief overview of the Android OS itself and its security features. In order to understand how to properly secure Android applications using SE Android, we must first have a thorough understanding the properties (permissions, IPC, etc.) of applications in the Android environment. We will also indicate weaknesses in the features and, in some cases, examples of exploits.

A. ANDROID FRAMEWORK

The Android Framework is laid out in Figure 2. As stated earlier, Android is based on the Linux kernel that provides the main system services to the Android software stack. The Linux kernel sits at the lowest level. It contains services such as device drivers, networking, and management of the file system, memory, power control, and process creation and management [18]. It also includes `init`. `Init` is the process responsible for taking care of all initialization when Android is booted up. `Init` processes the `init.rc` script file to set up the native services and performs the functions of a typical Linux system boot up.

Several kernel enhancements, such as the `binder` driver, discussed below, were added to support Android.

The next level up in the framework contains the Android native libraries. Written in C and C++ these libraries are used by numerous system components in the application layers. The Android Runtime includes the Dalvik virtual machine and core libraries.

All applications in Android run in a Dalvik VM. Android applications are written in Java which must first be converted into Dalvik executable files in order to run in the Dalvik VM. One may think that this provides sandboxing of applications, but the Dalvik VM is not a security boundary as any application can also include native code.

The application framework layer provides tools and services for use by applications. Like the applications themselves, these are all written in Java. There are a few critical system processes in Android to make note of [19].

One of the first services started is `zygote`. Aptly named, `zygote` is the master Dalvik VM process and the father of all Dalvik processes on the device. All processes are forked from `zygote`. Along with `zygote`, `init` also starts up some daemons. For instance, `vold`, the volume daemon, which manages the file system and `rild`, the radio interface link daemon, are started.

The next process to start up is the `system_server`. The `system_server` manages many of the native services. It is responsible for initializing most of the core services. Among these are:

- **activity manager -** responsible for managing the activities running on the device. It manages memory and state information as activities move through the activity stack. It is also the entity responsible for performing permission checks on `intent` deliveries [41].
- **system content providers -** a number of content providers are provided with Android for managing contacts, photos, music, etc.

The activity manager starts several core applications: `com.android.phone` and `android.process.acore`. At this point most of the services are up and running.

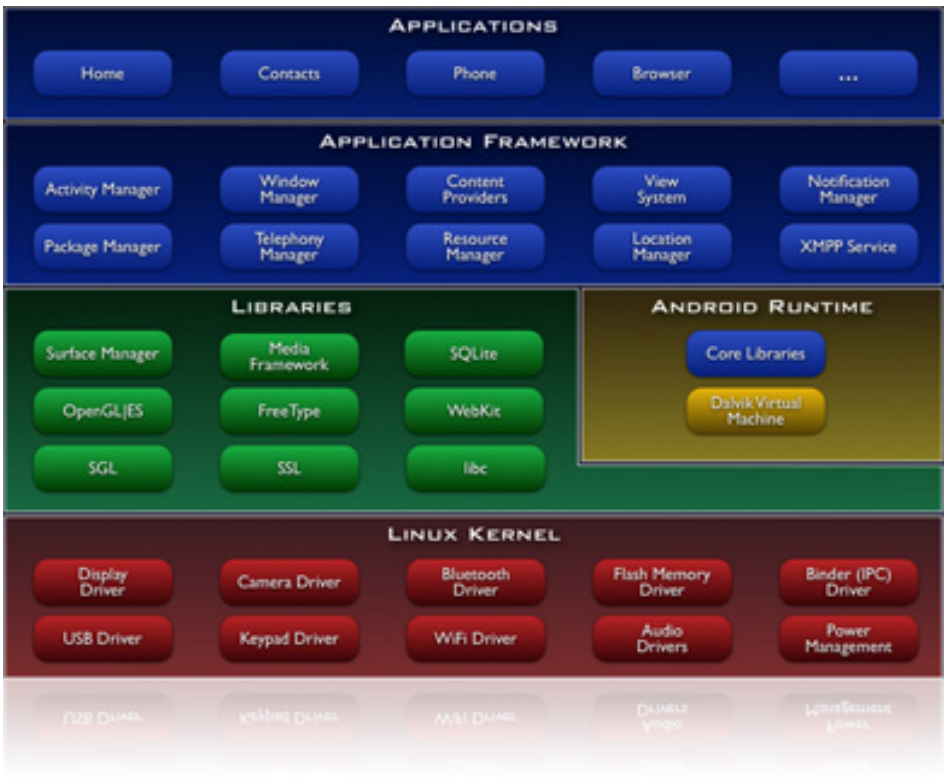

Figure 2. Android Framework, from [20]

B. ANDROID SECURITY MODEL

Android is a very interesting OS in terms of its security model. It is unique in that it attempts to provide the users more control of the security of the system. This, combined with its open source nature, leads to some interesting techniques and challenges for providing security.

Android is largely based on Linux. As such, it has retained some of the security features familiar to Linux users including things like UNIX user identifiers (UIDs) and file permissions, which it utilizes for DAC [21].

There are two primary features of the base Android security model. The first is Application-level permissions that encompass controls of application components and system resources. The second is Kernel-level sandboxing, which involves isolating applications from each other and the system, and also prevents the bypassing of application-level security controls.

Perhaps the most important principle in the Android security model is the concept of sandboxing. Android applications run under their own unique UID with their own permissions with one exception. All applications must be signed by the developer's private key when placed on the Android market. This signing is used to "identify the author of an application and establishes trust relationships between applications" [18]. The system allows for different applications to run under the same UID if signed by the same author.

By default, applications running under one UID can neither read nor write data of applications signed by a different UID. Sharing data in Android must be done explicitly through the use of the inter-process communication (IPC) techniques. This isolation provides great inherent security because compromised applications do not necessarily result in a full compromise of the device provided the developer is careful when declaring permissions and the user correctly validates application permission requests on installation.

However, even with such security features, Android is still susceptible to malware. This weakness comes from two primary sources: unintended installation of malicious applications and weak security practices by application developers. Malicious applications can vary significantly in their maliciousness. Unprivileged malware, or malware that has access to `Normal` permissions, as discussed in the next section, could be a great annoyance and hindrance to the user by randomly playing noises or running down the battery. Privileged malware, or malware with `Dangerous` permissions, can lead to stolen data or monetary loss. The greater danger comes from legitimate applications that have been coded carelessly leaving data or services unsecured. Accordingly, developers need to take greater care in considering how they are using users' data and what services they are providing.

Several instances of careless data management in Android (and other systems) applications were documented in late 2010 [22]. It was discovered that several banking applications for both Android and the iPhone were storing user information on the devices in an insecure manner. The USAA android application stored images of pages visited using the application that could potentially contain sensitive information. TD

Ameritrade's application was storing usernames in plain text on the phone. Bank of America's application saved the user's answer to their security questions in plain text on the device.

C. ANDROID PERMISSIONS

Every application that a user installs comes with a request for a set of application-specific permissions that is set by the applications developer. These permissions allow the application to access system (or other application's) data or services. These include things like `READ_CONTACTS`, which grants permission to read the user's contact book and `SEND_SMS`, which allows the application to send SMS messages. Additionally, custom permissions may be created and used by the application. For instance, an application could use a custom permission to restrict other applications from using its service. Permissions for each application are found in their individual `AndroidManifest.xml` files created by the developer and placed on the mobile device as part of the installation process. These permissions are displayed to the user who must agree to them in order to install the application. An example of this process is shown in Figure 3.

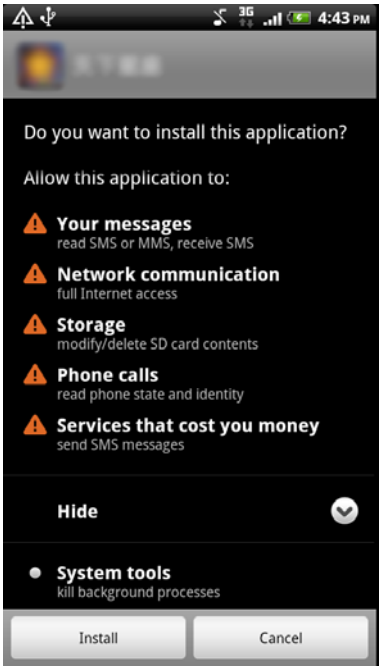

Figure 3. Permission Request, from [23]

There are four protection levels for permissions:

Normal – Permissions for minor features like `VIBRATE`.

Dangerous – Permissions for features that can reconfigure the device or incur fees. Users will be explicitly warned about dangerous permissions on install.

Signature – The permissions are only granted to other applications signed with the same key as this program.

SignatureOrSystem – These permissions are for programs installed as part of the system image and typically aren't be used by developers.

This system of requesting and granting permission puts a great deal of the responsibility for security in the hands of both the developer and user. The user must be aware of both what the application is advertising it does and the permissions it requests. Fortunately, the application reviews on the Android Market and developer's reputation may help alert naïve users who attempt to install malicious or insecure applications on their devices.

The developer must be sure to follow the principle of least privilege (PLOP) when requesting permissions. The principle of least privilege is a common concept in information security that requires every entity (be it a process, user, or program) have access only to the information and resources that are sufficient and necessary for its purpose [4]. Unfortunately, developers do not always follow this principle. As found in [24], as many as one-third of Android applications, as configured in their manifest, are over privileged. However, typically only a few permissions were over-privileged so there appears to be a good faith effort in the developer community to follow the principle of least privilege.

Once an application is installed, the permissions can't be changed. This eliminates direct attempts at privilege escalation (attempts that are confined to a single application). Privilege escalation can still occur through indirect means.

How can attackers conduct privilege escalation attacks indirectly? One concern is that of application collusion attacks. Put simply, a collusion attack occurs when two

different permission sets combine to offer actions not intended to be allowed. In the Android world, this can be accomplished by two applications communicating.

Imagine that an attacker writes up two applications: one for managing contacts, and another for a cloud-based calendar application. The contacts manager has permissions to read from the contact list and the calendar application can connect to the network. Assume that these applications are following POLP. On their own, these applications can be considered secure. However, if the contact managing application can communicate, overtly or covertly, with the calendar application, then there is the potential for contacts to leak out over the network to an attacker.

D. APPLICATION COMPONENTS

Android applications can be made up of four primary components: activities, services, content providers, and broadcast receivers. This section will briefly describe these components and their related protection mechanisms.

1. Activities

An Android application is a collection of tasks, each of which is called an activity. Each activity within an application has a unique task or purpose. They are the components with which users directly interact. Android allows for multiple applications to run concurrently, but there is only one activity actively running at any given time. The Android OS keeps track of all running activities on an activity stack. The activity on top of the stack is the active activity, while those below cannot be interacted with until all activities higher on the stack are destroyed.

Activities are started via `intents`, which are discussed later. The Activities are usually run in their own distinct process so they cannot access the calling process's data. Control on who is allowed to start a specific Activity is provided by permission checks in the application manifest; specifically by adding the `android:permission` attribute in the `<activity>` tag.

2. Services

Tasks that do not require user interaction can be encapsulated in a service. Services can be used in a number of ways: offloading time-consuming processing, performing a task that needs to be done regularly, or, as the name implies, providing a service for other components. Like Activities, they can be started with an `intent`. And, similarly, they can be protected via permissions in the `<service>` declaration in the manifest. Applications can communicate with services using the `bindService()` method that will result in a communications channel called a binder channel (discussed later).

3. Content Providers

Content providers are essentially databases and the programs to access them used both for data storage and the sharing of information among applications. They are SQLite databases [25]. Android provides a number of default content providers. For instance, the provider called `ContactsContract` contains the phones contacts. The browser provider manages the web browser history and bookmarks. The `MediaStore`, `CallLog`, and `settings` providers are also provided.

Access control to content providers is achieved through permissions. There are two separate permissions, read and write, with the write permission being a blind write. As with all permissions, these must be declared at application installation time and cannot be requested at run-time. Applications that wish to access another application's content provider must specify the permissions using the `<uses-permission>` element in their manifest.

The application that created and manages the content will of course always have full read and write permissions. If a content provider's application does not specify access permissions in its manifest, then no other application will have access to the provider's data. This allows for private data storage.

4. Broadcast Receivers

Broadcast messages are one way in which applications and components can communicate with each other. Broadcasts are sent out as `intents` to multiple applications. Broadcast receivers can read and handle these `intents`, allowing for messaging between applications. Which messages are received can be restricted by specifying the sender's manifest permissions. This will cause the activity manager to check if the sender has the required permission before delivering the `intent`.

Broadcasts can also be protected in the other direction, requiring that the receiver have the appropriate permissions declared in its manifest to receive the message. In this way the applications, which receive a particular intent, can be controlled.

E. INTENTS

`Intents` are the primary method by which Android processes move data and is the mechanism used for the majority of Android's inter-process communication. They are essentially data containers. Data sent using `intents` are actually sent over binder interfaces, the real backbone of Android IPC (which is discussed in the next section). As most developers deal at the `intent` level of abstraction, it is important to talk about `intents` specifically.

An `intent` generally consists of a recipient, an action to be performed by the recipient, and often data. If a recipient is explicitly identified, then it is sent to the specified recipient; if not, then it is up to the Android platform to determine which application can perform the requested action. There are a number of ways in which `intents` are used in Android. These include: starting a new activity, sending broadcast messages, communicating with services, accessing data in content providers, etc.

By default, applications can only receive system (internal) `intents`, they will not respond to `intents` sent by other applications. In order for an application to receive external `intents`, the receiving application's manifest must be configured appropriately. Either the application must have the `EXPORTED` flag set

(`android:exported="true"`), or the `intents` must be explicitly identified via Intent Filters.

Intent filters by themselves offer no security against hostile callers. To enable control of who can call a certain application component, a permission requirement can be added to the receiving application's manifest corresponding to the appropriate component. Activity permissions can restrict who starts the activity. Service permissions restrict who can start or use that particular service. Broadcast receiver permissions control which applications can receive the broadcast `intents`. Content provider permissions can control who can read or write data in the content provider.

As stated earlier, `intents` do not have to explicitly identify a recipient. This leads to a vulnerability to hijacking attacks. For instance, if an `intent` is sent to open a website without setting the recipient to `com.android.browser`, then it is possible for a malicious application to intercept this `intent` and launch a response to the request.

```
Intent i = new Intent(Intent.ACTION_VIEW)
i.setData(Uri.parse("http://www.google.com"))
this.startActivity(i)
```

Figure 4. Implicit `intent`

Another potential vulnerability is that of the confused deputy. In the confused deputy problem, an innocent but perhaps unsecure program is enticed by a third party into misusing the authority of the innocent program. Cross-site request forgery is a good example of a well-known confused deputy attack. In Android, this type of attack exploits the permission scheme [26]. In Figure 5, application A, the attacker application, has no permissions; application B, the confused deputy, has permissions to access application C, but no permission checks on its exported components. Application B is thus vulnerable to a confused deputy attack by allowing A access to its exported components in turn providing A access to the protected components in application C. Of course, this can be mitigated by B ensuring it conducts permission checks on its components, but in this case we put trust in the developers and applications, we are not relying on either Linux or Android mechanisms for protection.

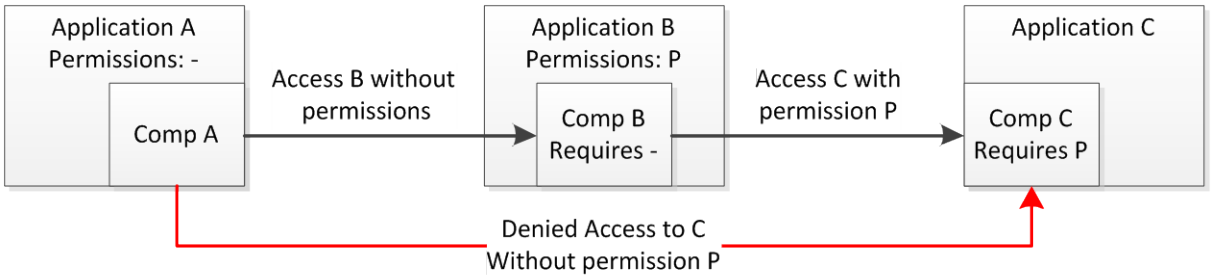

Figure 5. Confused Deputy Attack

F. BINDER

Binder is the framework for inter-process communication in Android. It is derived from OpenBinder developed by Palm Inc. [62]. It has been incorporated into Android to facilitate communication between Android applications. We will give a brief overview of how binder works. One thing to consider while reading this is that, binder can be considered a capability-based architecture.

Binder is a kernel module that allows for two applications to communicate. An application that wishes to provide a service for other applications must create a binder interface which the other applications will use to communicate with that particular service. When creating this interface, a binder reference token is created that uniquely identifies that binder interface. This token, along with the service's common name (that which is published in the manifest), is registered with the service manager. In order for other applications to communicate with the service, they must 'know' the token of the service with which they wish to communicate. 'Knowing' a token means that it has been added to a data structure that the binder kernel module maintains. The binder module creates and manages the data structures that hold all the binder references each application knows. These trees are populated by the binder kernel module as binder references are passed from one application to another.

If an application, A, wishes to communicate with another application, B, several steps must take place. First, A will ask the service manager what B's binder token is by providing B's common name. The service manager will provide A with B's token. Application A will then make a binder call with B's token, at which point the binder

kernel will check to make sure that B's token is in A's permission tree. If so, communication can proceed; if not it is halted. The purpose of this scheme is to ensure that binder references cannot be guessed. This, along with the uniqueness of binder tokens allows for the binder token to also be used as a shared key. The message sequence for a calling application wishing to use a service provided by another application is shown in Figure 6.

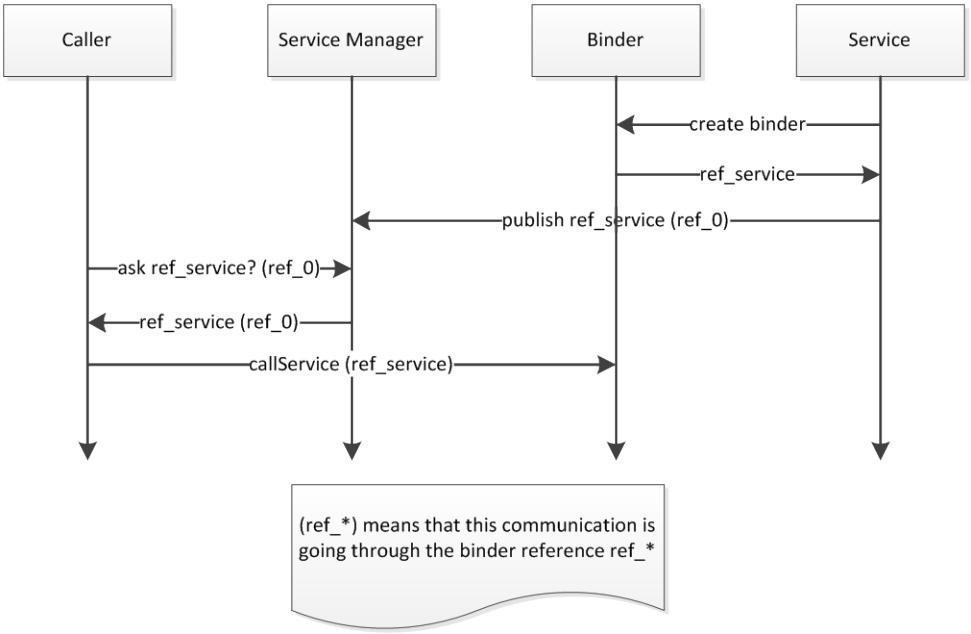

Figure 6. Example message sequence diagram for binder

Note that the application level permissions of the service and the application are not checked by binder. This means that the service manager must check the permissions of calling applications in order to maintain security. What the binder does to facilitate this is to ensure that the calling applications UID and PID are provided to the service; thus preventing spoofing of permission sets.

This should sound similar to capability-based systems. In the binder framework, the binder references are the capabilities. The only right afforded to applications with a particular reference, is the ability to use that reference to communicate with the

associated application. In the binder framework, the primary distributor of capabilities is the service manager. When a component wishes to be made public for communication it provides its capability to the service manager, essentially inviting anyone to communicate with it.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SE LINUX

A. INTRODUCTION

In this chapter, we will briefly introduce SE Linux and the basic principles behind it. There are entire books written on the subject of SE Linux, so we will only cover the basics [28].

B. SE LINUX ACCESS CONTROL MODELS

SE Linux uses a combination of MAC, Type Enforcement (TE), and Role-based Access Control (RBAC) for its security model. At the TE level, SE Linux associates each process with a domain. Every domain is given a set of permissions that is necessary and sufficient for it to function properly. These permissions limit the files the domain can access and the types of operations the domain can perform on the files. The files themselves must also be labeled with permissions. These permissions are called a file's security context. Security contexts are stored in a table and are identified by security identifiers, or SIDs. SE Linux makes its security decisions based on these SIDs. During system initialization, initial SIDs are loaded into the security context table.

In SE Linux, the default rule is to allow no access, so every access must be specified with a TE rule. This type of fine-grained access control leads to exceedingly large security policy configurations and makes analyzing them in their entirety somewhat difficult. SE Linux uses RBAC in addition to TE to help simplify user management. Additionally, SE Linux provides options for MLS.

1. Type Enforcement

Type Enforcement is the main model at work in SE Linux. Every subject and object in the system is given a domain or type label. Subjects are labeled with domains and objects are labeled with types. In SE Linux, domains are actually just types associated with processes. The term domain is used in order to avoid confusion. This also means that SE Linux only has one matrix that represents access control. Instead of having DDTs for domain-type access control and DITs for domain interactions, as in the original

Type Enforcement implementations, SE Linux uses a single matrix used by the Security Server. The Security Server makes the security relevant decisions based on the SE Linux policy.

In the TE model, access is denied unless an allow permission is explicitly defined in a TE rule. Figure 7 displays a typical type definition and rule declaration. Type declarations name a type, and, optionally, associate attributes with the type name. Rule declarations usually have four parts:

Source type – typically the domin type of the process attemphing access

Target type – the type of an object being accessed

Object class – the class of object that the specified access is permitted. These group objects of the same category. For instances files, directories, and sockets would be different object classes.

Permissions – the kind of access the source type is allowed for the target and object class.

```
type shadow_t, file_type;
allow auth shadow_t:file;
```

Figure 7. Example type definition and rule declaration, from [28]

In Figure 7, the type declaration is identifying `shadow_t` as a type and associating it with the attribute `file_type`. The rule declaration is allowing the auth domain to read or get attributes for a file object with a type of `shadow_t`.

The second statement in Figure 7 is an example of one of the two general categories of TE rules. Access Vector Rules (AVR) are rules that authorize access permissions for subjects over objects. There are a number of different AVRs: `allow`, `auditallow`, `dontaudit`, and `neverallow`. `Allow` rules permit access. `Auditallow` rules will allow access, but will also log the action. `Dontaudit` denies access, but does not log the attempted action. `Neverallow` specifies that the action should never be allowed, even if there is an allow rule somewhere else in the policy

allowing for it. Any action which does not have a corresponding rule is automatically denied and the attempt is logged.

The other category of TE rules are the transition rules. Transition rules define default types assigned during object creation or domain transitions. Newly created objects are assigned types based on the domain of the creating process and the object's class.

Domain transitions are domain changes caused by a process performing an *exec* command (new process creation). The new default type that is assigned to the *exec*'d process is based on the domain of the current proccess and the type of the program being executed. It's important to note that domain transition rules only specifiy that a given transition can occur; it does not allow it to occur. Therefore, domain transitions must also have an associated AVR to allow for the transition to occur.

```
type_transition initrc_t ping_exec_t:process ping_t;
     allow initrc t ping t: process transition;
```

Figure 8. The first rule specifies a transition from `initrc_t` to `ping_t` on execution of a `ping_exect_t` process. The second rule allows that transition to occur, from [28]

2. Role-based Access Control

RBAC works alongside TE in SE Linux. For operations to be allowed in SE Linux, they must satisfy both TE and RBAC constraints in addition to standard Linux DAC. RBAC in SE Linux is essentially the same as described in Chapter II. The three basic elements are:

- users are authorized for a set of roles
- transitions between roles are authorized, but only if a transition between the two roles in question is authorized
- every role is authorized for a set of domains.

Each of these three elements is governed by different statements.

Role assignments are declared in user statements:

```
user u roles { sysadm_r };
```

Figure 9. The user `u` is authorized for the role `sysadm_r`, from [28]

Role transitions are determined by allow statements:

```
allow sysadm_r student_r;
```

Figure 10. A process running in the `sysadm_r` role can transition to the `student_r` role, from [28]

And lastly, authorizing roles for particular domains is accomplished using role statements:

```
role sysadm_r types ifconfig_t;
```

Figure 11. Role `sysadm_r` is authorized to enter the `ifconfig_t` domain, from [28]

3. Multi-Level Security

In addition to TE and RBAC, SE Linux also supports MLS. Security contexts can be given a level or range that corresponds to its security level. These levels should form a lattice [29]. SE Linux utilizes constraints to enable the enforcement of properties such as no read-up and no write-down.

C. SE LINUX POLICIES

SE Linux policies are typically compiled into binary policy files from a number of different source files. Originally, policies were monolithic, meaning the entire policy was contained within a single file. However, policies quickly grew to the point where they contained tens of thousands of lines and thus were very difficult to work with. The increasingly complex nature of SE Linux policies made modularization a necessity [28]. Now, there are four basic types of sources files used to describe SE Linux policies:

- Standard source files define things like domains, types, file contexts, and macros.

- Configuration files are typically modified by administrators to define users and roles.
- TE files define the policy for a particular domain.
- File context files are used for the initial labeling of objects.

Many user defined abbreviations and definitions are used to simplify and make the configuration files more readable. These are typically defined in terms of "macros." When creating the binary policy files, all of these files are combined and run through the Linux Gnu m4 macro processor to generate a `policy.conf` file [61]. The m4 macro processor is simply there to interpret and expand the macros in the policy files. The `policy.conf` file is then compiled into a binary `policy.VERSION` file.

D. SE LINUX POLICY TOOLS

As described above, SE Linux policies for systems grow in complexity very quickly, making it difficult for security administrators to create, use, and understand the security posture of a system described by these files. A number of tools have been developed to assist in analyzing SE Linux Policies [30–32]. In our research we used two tools:

Apol – Apol is one of the tools included in the SETools suite provided by Tresys Technology [32]. The suite contains tools for SE policy creation, management, and analysis. Apol provides a graphical interface for analyzing SE Linux policy files. It provides the ability to browse and search through components of the policy including types, attributes, roles, etc. It also allows for information flow analysis, domain transition analysis, relabeling analysis, and type relationship analysis. Figure 12 is a screenshot of Apol's transitive flow analysis feature.

Qisaq - Qisaq is a tool that was provided to us by the I4221 group at NSA for evaluation. Essentially, Qisaq is a Python interface to SETools. Qisaq was developed in house and is not currently available for public distribution. Because it has a Python programming language interface, it is easy to create scripts for analyses.

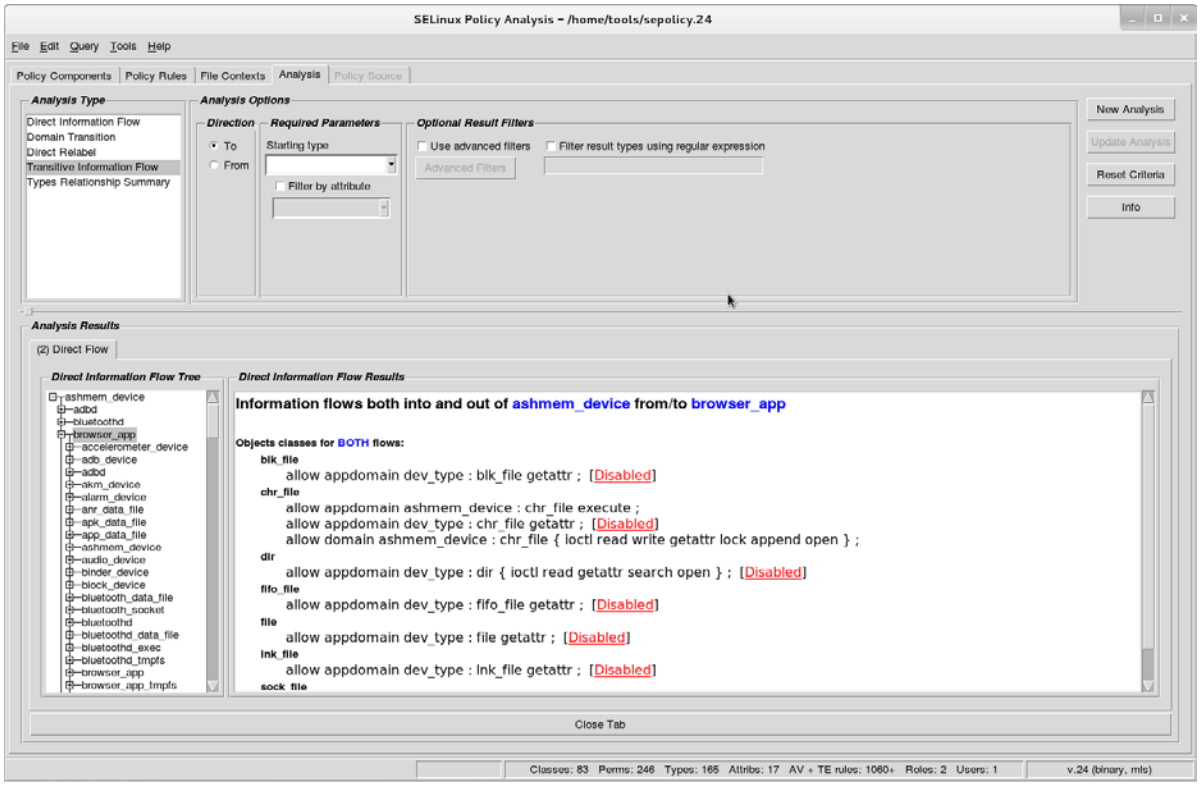

Figure 12. Apol Screenshot

V. SE LINUX IN ANDROID (SE ANDROID)

A. INTRODUCTION

Security Enhanced Android is under development by the NSA [12]. It was first released back in January of 2012. The goal of SE Android is to increase the security of Android by enforcing separation guarantees between applications. Since SE Android is still a work in progress, it has not been evaluated or approved for official use. While SE Android is intended to be a mainstream enhancement to Android with limited user interaction, we thought it would be interesting to see how it could be used to help secure a specific application architecture.

B. FEATURES

SE Android improves Android security in a number of ways. It confines privileged daemons, implements strong separation between applications, and provides for a centralized security policy that can be formally analyzed [12]. SE Android does more than just integrate SE Linux into Android. It also provides some extensions to the Android middleware to enhance MAC security.

Install-time MAC enables the checking of the permissions granted to installed applications against a policy. In other words, it can protect naïve users from installing applications with dangerous permissions by forcing a check against a MAC permission policy.

The following two mechanisms were not implemented in the version of SE Android used in this thesis [12]. The first is a tag propagation mechanism that tracks permissions between communicating application. This mechanism is like that provided by TaintDroid[4]. Initially, the set of tags for each application is based on its granted permissions. Upon IPC with another application, each application receives the union of the individual sets of tags from all the apps involved in the IPC. If that union violates policy rules, then the communication is blocked. This mechanism is an effective way to

[4] See the Related Works section of Chapter II.

prevent the application collusion attacks talked about earlier. The second unimplemented mechanism supports permission revocation. This allows the revocation of permissions from installed applications using the SE Manager application.

C. REFERENCE POLICY

SE policies for SE Android are found under the `external/sepolicy` directory in the SE Android build source files. These files can be downloaded directly from [33]. This section will highlight the capabilities of the reference policy that came packaged with the SE Android distribution. The version of the policy described here was downloaded as part of the SE Android modifications for Android 4.0.4. According to Smalley in [34], the "goals for the SE Android policy are to confine the privileged daemons in Android, ensure that the Android middleware components cannot be bypassed, and ensure that applications are truly isolated from one another at the kernel layer."

It is important to note that the build and policy we will be using is from July 2012. Since SE Android is a work in progress, the policy is continually changing as problems are discovered and new features are added.

1. Domain Rules

The basic domain rules are found in the `domain.te` file. This file contains the AVRs which all domains need to function. These access rules are all associated with the attribute domain.

The AVRs include intra-domain accesses that simply allow a domain to access its own elements like file descriptors, directories, etc. Device accesses are also included. Figure 13 displays the defined device accesses for every domain. The accesses granted to the domain attribute are kept to a bare minimum. For instance, the rule for `socket_device` only allows for searching of current sockets on `/dev/socket` and not creation of or reading from sockets.

```
allow domain device:dir search;
allow domain devpts:dir search;
allow domain device:file read;
allow domain socket_device:dir search;
allow domain null_device:chr_file rw_file_perms;
allow domain zero_device:chr_file r_file_perms;
allow domain ashmem_device:chr_file rw_file_perms;
allow domain binder_device:chr_file rw_file_perms;
allow domain ptmx_device:chr_file rw_file_perms;
allow domain powervr_device:chr_file rw_file_perms;
allow domain log_device:dir search;
allow domain log_device:chr_file w_file_perms;
allow domain nv_device:chr_file rw_file_perms;
allow domain alarm_device:chr_file r_file_perms;
allow domain urandom_device:chr_file r_file_perms;
```

Figure 13. AVRs specifying domain access rules for system devices, from [33]

2. Application Domains

The file defining AVRs for applications is found in `app.te`. These rules apply to applications that lack a predefined platform UID (system, radio, etc.). Originally, it was divided into three sections with rules for the following: `appdomain`, `trusted_app`, and `untrusted_app`.

The `trusted_app` domain is no longer in use in current versions of SE Android. It has been replaced in favor of more distinct trusted domains. It has been separated into four separate domains each with their own set of AVRs. They are the `platform_app`, `media_app`, `shared_app`, and `release_app` domains that correspond to the AOSP[5] build keys [36].

All applications are members of the `appdomain`. It provides the base set of rules that all applications must have in order to function. There are a few rules regarding its interaction with `zygote`. For instance, `appdomain` can use open file descriptors that were inherited from `zygote`. There is also a pair of rules that allow members of the `appdomain` to create file and directory permissions to enable sandboxing of their own

[5] Android Open Source Project maintains the source code for Android and its various versions.

files. Lastly, the appdomain uses a set of macros that describe binder accesses to allow for binder IPC use. These macros will be discussed in more detail in a later section, they allow the appdomain to perform binder IPC calls to members of the binderservicedomain and trusted_app domain.

```
allow appdomain zygote:fd use;
allow appdomain zygote_tmpfs:file read;
allow appdomain zygote:process sigchld;
allow appdomain system:fifo_file rw_file_perms;
allow appdomain app_data_file:dir create_dir_perms;
allow appdomain app_data_file:notdevfile_class_set
     create_file_perms;
allow appdomain system_data_file:dir r_dir_perms;
binder_use(appdomain)
binder_call(appdomain, binderservicedomain)
binder_transfer(appdomain, binderservicedomain)
binder_call(appdomain, trusted_app)
binder_transfer(appdomain, trusted_app)
```

Figure 14. AVRs for appdomain, from [33]

Since all applications are members of the appdomain, the policy divides applications into two categories: trusted and untrusted. Untrusted applications, naturally, have fewer permissions. Access is restricted to the network, Bluetooth, SD Card, as well as native applications. These are controlled by Booleans that are currently managed by the SE Manager application. Untrusted applications fall under the same rules as the appdomain regarding the use of binder.

The platform_app, media_app, shared_app and release_app domains were all originally united under the trusted_app domain. This was changed when install-time MAC was introduced to allow for more granularity in the permissions of the different system applications. All of these applications are also mlstrustedsubjects. This term will be discussed in the MLS section below.

3. Seapp_contexts

The policy contains a file called `seapp_contexts`, which is unique to the SE Linux implementation in Android. The `seapp_contexts` file is used to label, or add security contexts to application processes and their package directories. In order to label an application process, a domain must be specified. Similarly, to label an application directory a type must be specified. The `seapp_contexts` file is read by `libselinux/android.c` which labels entities based on a set of precedence rules. The rules are as follows:

(1) `isSystemServer=true`

(2) specified user before unspecified user

(3) fixed user string before user prefix string with a wildcard (*)

(4) longer user prefix over shorter prefix

(5) specified `seinfo` before unspecified `seinfo`

(6) specified name before unspecified name

This first check is whether the process is the System Server. If so, it will be labeled with the system domain. Note that there is only one System Server. The second check serves to determine if the application runs under a predefined platform user: system, nfc, radio, etc. The third and fourth checks simply deal with different user strings, with fixed strings coming before prefix strings (app_*) and longer prefixes coming before shorter ones. The fifth rule checks for an seinfo string that is used to look up the security context of an application process. The last rule checks whether a package name has been specified. This allows for specific application labeling.

```
isSystemServer=true domain=system
user=system domain=system_app type=system_data_file
user=nfc domain=nfc type=nfc_data_file
user=radio domain=radio type=radio_data_file
user=app_* domain=untrusted_app type=app_data_file
     levelFromUid=true
user=app_* seinfo=systemApp domain=trusted_app
     levelFromUid=true
user=app_* seinfo=systemApp name=com.android.browser
     domain=browser_app levelFromUid=true
```

Figure 15. `seapp_contexts` statements, from [33]

Figure 15 shows the statements from an early version of the `seapp_contexts` file. The first statement is simply assigning the System Server to the system domain. The second statement will map application processes under the system user id to the system domain and their application directories to `system_data_file`. Similarly, applications under the nfc and radio user id are given their own process and directory labels. The next rule for `untrusted_apps` is the catchall statement. Any application that does not match any of the other statements falls under the `untrusted_app` domain. The next statement is for `trusted_apps` and the last is specific for the application `com.android.browser`.

The `seinfo` string was rather unclear as to its purpose in earlier builds of SE Android. This is because it was hardcoded with only a `systemApp` value, which specified applications in the system partition. So, in Figure 15, any application in the system partition was labeled under the `trusted_app` domain with the exception of `com.android.browser` due to name precedence. This has since been updated with the introduction of install-time MAC and is explained in the next section.

4. Install-time MAC

Install-time MAC is a feature unique to SE Android. It allows for specification of MAC on the Android permissions that applications request at install time. The policy configuration for this feature is found in the `mac_permissions.xml` file. With it, MAC rules can be placed on the different permissions to either allow or deny a requested

permission. This allows for the Android permission security mechanism to be controlled by a central policy rather than individual users. Dangerous permissions can be explicitly denied so that naïve users are less likely to compromise their device. Figure 16 shows how the `mac_permissions.xml` file is organized.

```
<signer signature="308204a8308…" >
    <allow-all />
    <seinfo value="platform" />
    </signer>

<signer signature="308204a83082039…” >
  <seinfo value="release" />
    <deny-permission name="android.permission.BRICK"/>
    <deny-permission name="android.permission.READ_LOGS"/>
    …
    <package name="com.android.browser" >
      <allow-permission name="android.permission.
       ACCESS_COARSE_LOCATION”/>
      <allow-permission name="android.permission.
       ACCESS_DOWNLOAD_MANAGER"/>
      <allow-permission name="android.permission.
       ACCESS_FINE_LOCATION"/>
      …
    </package>
</signer>

<default>
    <seinfo value="default"/>
    <deny-permission name="android.permission.
     ACCESS_COARSE_LOCATION"/>
    <deny-permission name="android.permission.
     ACCESS_FINE_LOCATION"/>
    <deny-permission name="android.permission.
     AUTHENTICATE_ACCOUNTS"/>
    <deny-permission name="android.permission.
     CALL_PHONE"/>
    <deny-permission name="android.permission.CAMERA"/>
    <deny-permission name="android.permission.READ_LOGS"/>
    <deny-permission name="android.permission.
     WRITE_EXTERNAL_STORAGE"/>
</default>
```

Figure 16. mac_permissions.xml, from [33]

In Figure16, there are different sets of permissions for three different types of applications. The <signer> element determines what applications receive its permission set. Specifically, they represent the keys used to identify the authors of an application. The first signer corresponds to platform applications and the second signer corresponds to release applications. Generally, third party applications have their own unique key that must be added to this file to enforce MAC on permissions specific to that application. However, keys that are not specified in the mac_permissions.xml file fall under the default group.

Each <signer> element and the <default> element can have several child elements. The <seinfo> element corresponds to the seinfo string found in the seapp_contexts file. This will associate the application(s) specified by the <signer> element to the corresponding labeling rules in seap_contexts. This allows for controllable application specific labeling.

The <allow-permission> and <deny-permission> tags allow for white-listing and black-listing of android permissions. For instance, <deny-permissions name="android.permission.READ_LOGS"/> will deny the parent <signer> application the READ_LOGS permission. There is also an <allow-all> tag if an application should not be denied any permissions.

Lastly, the <package> tag can define the same rules for specific packages. The rules for the <package> element override those of its parent <signer> element. The <package> tag can also exist outside of a <signer> parent element in which case they can also have their own <seinfo> tag.

5. Important System Applications

In this section we will describe the policies for some of the important processes in Android. The functionality of some of the processes have been modified from their SE Linux versions for compatibility with SE Android.

a. General System Apps

The `system.te` file contains AVRs for the `system_server` as well as other applications that run under the system UID. The `system_server` is more privileged than the other system applications because, as noted above, it is responsible for managing native services. As such, it has permissions you would expect for service management: scheduling and killing processes, communicating with daemons, managing data and cache files, and a few other managerial operations.

The other applications running under the system UID are not as privileged as the `system_server`. These applications, which include the UI and settings applications, are given the basic set of permissions associated with applications. Additionally, they can perform binder functions for services and applications referenced in the `appdomain`. The settings application was originally responsible for managing the SE Linux mode (enforcing or permissive) and SE Linux Booleans. In current versions of SE Android, these are now managed by a separate system application called SE Manager.

b. Init

`Init` is an important daemon as discussed in the Android framework section. It is responsible for the initialization of Android. The `init` domain, along with the `kernel` and `su` domains, makes use of the `unconfined_domain` macro, which associates with it the type attributes `mlstrustedsubject` and `unconfineddomain`. The `mlstrustedsubject` exempts the domain from MLS constraints, and `unconfineddomain` allows it to do anything.

Also, since `init` is responsible for starting up the various daemons, it must allow for those daemons to transition from `init`'s domain to their respective domains. The macro `init_daemon_domain(domain)` sets the "permissions" in `te_macros` file that allow those transition to happen.

c. Zygote

`Zygote` is the process from which all other processes are spawned. `Zygote` has been modified in SE Android to allow it to set security contexts for the applications it spawns. It also maps the DAC credentials of its children to the security context [34]. As such, it must have permissions enabling it to perform the proper functions when forking new processes. It must also be able to transition into new domains. `Zygote` can transition into either the `system` domain or the `appdomain` domain.

```
allow zygote system:process dyntransition;
allow zygote appdomain:process dyntransition;
```

Figure 17. `Zygote dyntransition` permissions, from [33]

d. Service Manager

The service manager, as discussed earlier in the binder section, handles binder requests and transfers references. The service manager does not need to pass its own reference as it is static and known by all, and it only ever receives requests. Therefore, the service manager's rule set is simple.

```
allow servicemanager self:binder set_context_mgr;
allow servicemanager domain:binder;
```

Figure 18. Service manager IPC rules, from [33]

e. Media Server

Permissions for the media server are found in `mediaserver.te`. The media server is responsible for managing and indexing images, videos, and music files. It requires access to multimedia devices including: SD Card, camera, video, and audio devices. It is also a member of `binderservicedomain`, which marks it as being a binder service and allowing binder IPC to system services.

f. Installd

The installer daemon, like `zygote`, has been modified for SE Android. As the name implies, `installd` is responsible for installing applications. It has been modified to label the application data directories that it creates as part of the install process. It reads from the `seapp_contexts` configuration for labeling purposes.

6. Macros

This section will briefly cover some of the macros used in policy definitions. These macros provide shortcuts to assigning domains attributes or AVRs. The ones discussed here are those used for application policies, not those that are used for system applications. Most of these macros deal with control of IPC mechanisms.

The first macro we describe is the `app_domain(domain)` macro. This does two things. First, when it is expanded, it assigns the type attribute `appdomain` to the input argument domain. As mentioned earlier, the `appdomain` is associated with the base set of permissions required by all applications. The second part of the macro is actually another macro: `tmpfs_domain(domain)`.

The `tmpfs_domain` macro, when expanded, defines a unique file type for the domain to use when creating tmpfs (temporary file storage), shmem (shared memory), and ashmem (anonymous shared memory) files. It describes how to provide further guarantees of isolation for application data. Figure 19 shows these two macros. Note that the $1 signifies the first input parameter. If written as a function it would look like `app_domain($1)`.

```
define(`app_domain', `
     typeattribute $1 appdomain;
     tmpfs_domain($1) ')
define(`tmpfs_domain', `
     type $1_tmpfs, file_type;
     type_transition $1 tmpfs:file $tempfs;
     allow $1 $1_tmpfs:file {read execute execmod}; ')
```

Figure 19. `app_domain` and `tmpfs_domain` macros, from [33]

The `net_domain(domain)` macro associates the input domain with the type attribute `netdomain`. The AVRs for `netdomain` are found in the `net.te` file. These rules allow for the use of network sockets and connecting and binding to tcp and udp sockets.

```
allow netdomain self:{ tcp_socket udp_socket } *;
allow netdomain node_type:{ tcp_socket udp_socket }
node_bind;
allow netdomain port_type:udp_socket name_bind;
allow netdomain port_type:tcp_socket name_bind;
allow netdomain self:netlink_route_socket {create bind
read nlmsg_read};
unix_socket_connect(netdomain, dnsproxyd, netd)
```

Figure 20. `netdomain` AVRs, from [33]

When expanded the next pair of macros set up the AVRs for access control for general socket usage. As seen in the last statement in Figure 22, the macro, `unix_socket_connect( clientdomain, socket, serverdomain)`, allows for socket connections from a client domain to server domain via a socket. The other macro is

`unix_socket_send(clientdomain, socket, serverdomain)`.

This allows for the socket operation send to be performed from the client domain to the server domain via a socket.

```
define(`unix_socket_connect', `
     allow $1 $2_socket:sock_file write;
     allow $1 $3:unix_stream_socket connectto; ')
define(`unix_socket_send', `
     allow $1 $2_socket:sock_file write;
     allow $1 $3:unix_dgram_socket sendto; ')
```

Figure 21. Socket macros, from [33]

The last set of macros set up the AVRs for the controlled use of the binder. The first of these is `binder_use(domain)`, which sets up the domain to allow it to use

binder IPC. This entails allowing the domain to retrieve binder references from the service manager. Additionally, it allows sending and receiving binder references to and from itself. The transferring of binder references is accomplished by the macro `binder_transfer( clientdomain, serverdomain)`. The `binder_call(clientdomain, serverdomain)` macro allows for the client domain to perform binder IPC calls to the server domain. This is accomplished in two rules: allowing the receipt and calling of the server's binder reference and allowing the use of the servers file descriptors.

```
define(`binder_use', `
     allow $1 servicemanager:binder call;
     allow $1 self:binder { transfer receive };
     allow $1 ashmem_device:chr_file execute; ‘)
define(`binder_call', `
     allow $1 $2:binder { receive call };
     allow $1 $2:fd use; ‘)
define(`binder_transfer', `
     allow $1 $2:binder transfer; ')
```

Figure 22. Binder macros, from [33]

These are the macros of most interest for writing policies for specific applications. There are many other macros found in the `global_macros`, `te_macros`, and `mls_macros` files. Some of the macros for MLS will be discussed in the next section.

7. MLS

There are two files relating to MLS in the policy: `mls` and `mls_macros`. These files provide the architecture that allows for MLS policies to be used. Currently, there is only one sensitivity level defined: `s0`. As mentioned earlier, SE Android improves application separation by using MLS categories. Every application is labeled with its own category: `c0` to `cN-1`.

The `mls` and `mls_macros` files provide for easy deployment of MLS policies. They provide for placing constraints on: processes, sockets, directories and files, and IPC. These constraints follow the Bell-LaPadula model with the exception that

`mlstrustedsubjects` are exempt from these constraints. There are 13 important system processes that are labeled `mlstrustedsubjects`: `adbd`, `debuggerd`, `drmserver`, `init`, `installd`, `kernel`, `mediaserver`, `netd`, `su`, `surfaceflinger`, `system`, `vold`, and `zygote`.

D. SE MANAGER

SE Manager is an application packaged with SE Android that allows for the configuring of SE Android enforcing modes: SE Linux enforcing mode, and MAC enforcing mode. The SE Linux mode corresponds to the traditional SE Linux enforcing and permissive modes. Enforcing mode actively denies AVR violations. Permissive mode monitors for and reports violations but does not deny actions. The MAC enforcing mode is for the enforcing of the install-time MAC permission checks. The application also allows for the setting of the various Booleans included in the policy. These Booleans allow for conditional granting of access based on their value. For instance, the `android_cts` Boolean will grant the permissions needed for the Android Compatibility Test Suite to run when set. Lastly, the application allows for the viewing of SE Linux and MAC logs.

E. SE ANDROID VS EXPLOITS

In [10], Smalley outlines how various Android exploits would be prevented by SE Android. Most of these exploits are simply variations of the same vulnerabilities. In this section we will briefly describe several of these case studies.

1. RageAgainstTheCage

RageAgainstTheCage, also known as "CVE-2010-EASY," is an exploit that results in an `adb` (used for debugging) shell running as root [35]. It takes advantage of a vulnerability in the `setuid(uid)` function. The problem is that `setuid` does not drop privileges if the RLIMIT_NPROC resource limit is hit. The setuid(2) man page declares [36]: "**ERRORS - EAGAIN** The uid does not match the current uid and uid brings process over its **RLIMIT_NPROC** resource limit." The **RLIMIT_NPROC** is "the

maximum number of processes that can be created for the real user ID of the calling process" [37].

RageAgainstTheCage begins by iterating through the processes in the `/proc` directory, where the proc file system resides. That directory contains information about all the processes running in the system. It is searching for the `/proc/pid/cmdline` file belonging to `adb`. This exploit would be prevented by SE Android because `adb`'s `/proc/pid` directory has been labeled with the security context `u:r:adbd:s0`, which would render it unreadable by untrusted applications with the context `u:r:untrusted_app:s0:c`*`N`*. Furthermore, upon exec of a shell, the security context would change; so while the shell would indeed be running as root, it would be running in a restricted security context.

The Zimperlich exploit does essentially the same thing but spawns an application component with escalated privileges by causing `zygote`'s `setuid` to fail [35]. It too would not work in SE Android.

2. Exploid

Exploid is a jail-breaking exploit that appeared in 2010. It utilized a vulnerability in Netlink as detailed in CVE-2009–1185 [38]. The problem is that `udev`, the device manager, did not verify whether Netlink messages came from kernel space or user space. This allowed local users to gain privileges by sending a Netlink message from user space. In Android, the vulnerability was inherited as much of `udev`'s functionality was ported over.

Exploid works by using the Netlink vulnerability to perform an arbitrary write as root to an arbitrary file; in this case `/proc/sys/kernel/hotplug`. Hotplug gets invoked anytime a device is plugged in. The contents of hotplug are overwritten to point to the exploit code to be executed. In SE Android, this exploit would be stopped in two ways. First, the creation and use of Netlink sockets is denied by the policy. Secondly, the write to `/proc/sys/kernel/hotplug` would be denied because of the security context despite the writing process being root.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. PROOF OF CONCEPT APPLICATIONS AND POLICY

A. SCENARIO INTRODUCTION

The scenario that our proof-of-concept is based on is a calendar application that can work with data from calendars of different categories.

For this thesis, we built a system that will display data from two different calendars in non-interacting domains. The system provides a single display for the two calendars. There is a mechanism for managing one of the two calendars. The display will only show full information for the calendar that is currently being managed.

We did not code up the complete application due to its complexity and our time constraints. The goal was to determine how to configure the security policy to allow for their secure interactions and operations. Instead the applications are merely partial implementations that reflect the kinds of operations that would be performed in such an application. The communication channels and data stores are complete so that the policy can be properly analyzed.

Our testing environment is a 64-bit Fedora16 virtual machine that contains the Android ARM emulator running Android 4.0.4 (Ice Cream Sandwich) using NSA's release of SE Android pulled in June 2012 from [39].

B. ARCHITECTURE OF APPLICATIONS

The application is actually a set of applications that work together. There are three different types of applications: the main application, the trusted controller, and the separate calendar applications. A diagram of the components and their communications channel is shown in Figure 25. The source code for these applications can be found in Appendix A.

1. Main Application

The main application, shown in Appendix A, Section 1, is the only application that is visible to the user and that the user can interact with. This main application displays the data from the two separate data stores with one set being redacted depending

on which mode it is in. The main application does not communicate with these data stores directly. Instead it feeds requests through a trusted controller.

There are two activities in the main application. The first, seen in Figure 23, allows for the selection of which mode it is in. This mode determines which data is displayed and which is redacted.

Figure 23. Selection Activity of the Main Application

The second activity communicates with the trusted controller to perform actions on the calendar applications. It does so by binding to an IBinder interface exported by the trusted controller. An IBinder interface allows for an application to export functions as a service for other applications to use. In this implementation, the only operation available is a read table request which simply has the data sent to this application to display. Communications with the trusted controller is done over a binder interface to emulate how this would be done in a complete implementation. Lastly, the second activity implements a broadcast receiver to handle the data being sent to it by the two calendar applications. It would probably be better to use regular `intents` instead of broadcast `intents`, but broadcast `intents` were used to allow testing of the install-time MAC

feature of SE Android. After receiving that data, the activity then displays it in a table format as seen in Figure 24.

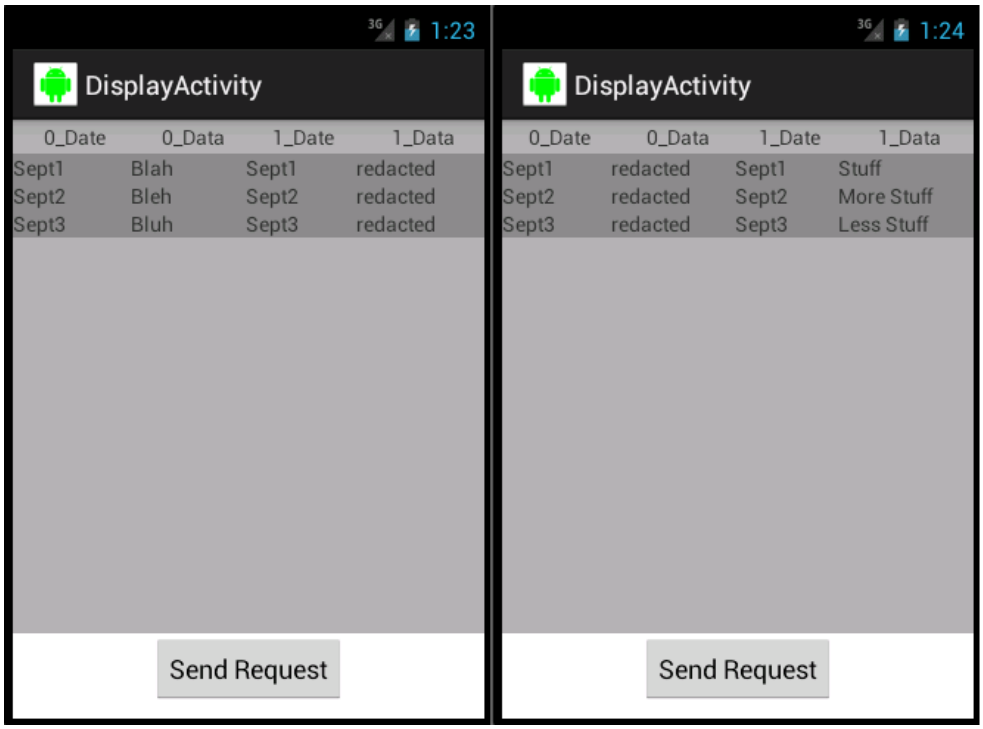

Figure 24. Display Activity with view 0 selected (left) and view 1 selected (right)

2. Trusted Controller

The trusted controller is the main reason we did not fully develop this application as its complexity exceeds the scope of this thesis. For the purpose of this thesis, we assume that the trusted application is 'secure', that is that it is correctly implemented. We are assuming that it will only send the correct request to the calendar app that it was instructed to. The trusted controller receives requests from the main application and then communicates with the separate calendar applications. These requests are received via an IBinder interface. Only one function was exported to this interface. That was the `readTables()` function, which would ask the calendar applications to send query results to the main application.

Communications with the calendar applications is also done via an IBinder interface connection. The reasoning behind this is that it is expected in normal usage of

the full application to have the user perform multiple requests that would make an established binder connection more efficient.

3. Calendar Applications

There are two calendar applications in our implementation. For the purposes of this thesis, these two applications will communicate with content providers on the SE Android device to store their data. In the real world, it is more likely that this data will be housed in the cloud for availability on multiple devices. In our proof-of-concept each calendar application implements its own content provider. Within each content provider the applications store 'appointment data' in a database. This database consists of one table, dates, with the following columns: `date_id`, `date`, and `comments`. This table is meant to represent the type of data found in a simple calendar application. The date represents the date/time of the appointment and the comments describe the appointment. For this study, these tables have been prepopulated with data to simplify the applications, i.e., we are not implementing the calendar update function. Inserting more data into the content providers of each of the applications would take the same communication paths as reading does, and so it would not affect the SE Android policy.

Like the trusted controller, these applications implement an IBinder interface. This interface implements one function that allows for queries against the content provider. The results of those queries are then packaged up in a broadcast `intent` and sent to the main application. The data sent to the main application is added as an extra data item to the `intents` using `intent.putExtra(name, data).`

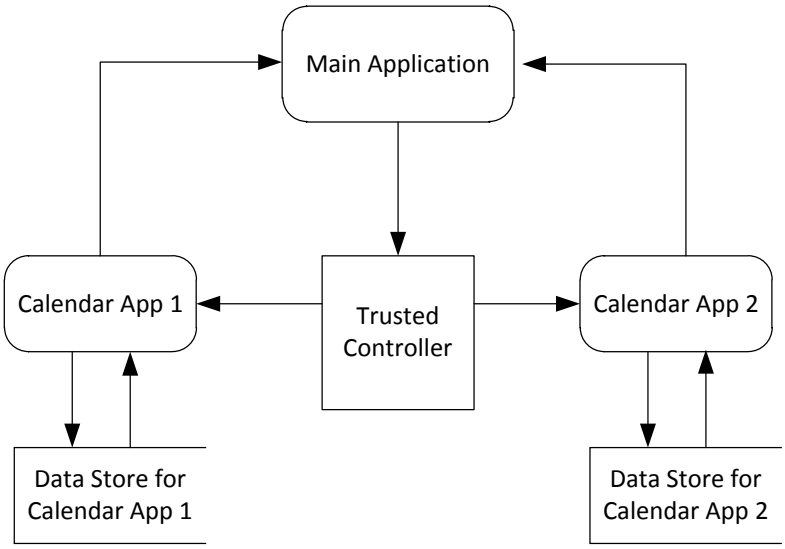

Figure 25. Communication channels for the applications

C. SECURITY GOALS/REQUIREMENTS

There are several security goals for this proof-of-concept. Most goals relate to the prevention of the leaking of information from one calendar to another and from the calendars to other applications. The goals are as follows:

- There is to be no information flow between the individual calendar applications.
- The data stored in the content providers should be accessible only to the proper applications.
- The data sent via broadcast `intents` should only be received by the main application.
- Only the applications specified earlier within the proof-of-concept are allowed to bind to the exported services.

One may notice that it appears possible for data to pass from one calendar application to the other through the allowed communications channels. For instance, the display application could receive data from one calendar application, and then send it to the trusted controller, which will then send it to the other calendar application. As noted earlier, this is not a complete implementation and we are trusting that the trusted controller would not allow that to happen. However, in the discussion section, we propose an alternate architecture to prevent this channel.

D. ANDROID SECURITY

Each application is protected by permissions declared in their AndroidManifest.xml file. Five custom permissions have been defined: one for access to the controller, two for access to each of the calendar apps, and two permissions to be used for the broadcast `intents`. The controller and calendar applications each define and place a permission on their services. The display application uses the controller application permission, and the controller application uses each of the calendar application permissions. The calendar applications place a permission requirement on the broadcast `intents` they send to the display application using `sendBroadcast(intent, permission)`. This ensures that only applications with the proper permission can receive that `intent`.

The content providers implemented by each of the calendar applications are protected by default. If no permissions are added to the manifest regarding access to the providers, then only the application implementing the providers can access it.

1. Deficiencies

The only place where the basic Android security features are weak is in the permissions. The problem lies in the fact that malicious developers can simply read the manifest of the application declaring the permission, and then use that permission in their own application. The user would then be asked whether to grant that permission to the malicious application on install. The `android:description` tag would be displayed and, if that tag was not defined or gives insufficient detail, a naïve user may grant it permission.

There are two ways that SE Android can mitigate this problem. The first is by placing MAC on the communication channels between the applications themselves. This will prevent any outside application from communicating with applications it should not have access to. The second way is by placing MAC on the permissions themselves. Using the install-time MAC feature, the permissions can only be granted to certain applications.

E. SE LINUX POLICY DEVELOPMENT

The process of writing policies for new applications is fairly straightforward for SE Android. This is because there are already macros and domains defined which provide most of the permissions needed for applications to work. There are only a handful of files that need to be updated: `seapp_contexts`, optionally the `mac_permissions.xml`, and the separate TE file containing the policy.

The policy for the proof-of-concept application was made by first labeling the applications via the `seapp_contexts` file. Then, with permissive mode enabled, the application was run looking for any logged SE Linux denials. Based on these denials, permissions were added to allow the application to function properly. This allowed for the POLP to be followed when adding AVRs to the policy. The contents of the files for the policy changes and additions can be found in Appendix B. We describe some of the details below.

1. App.te

Some changes were made in the `app.te` file. In earlier versions of SE Android, there were two types of applications defined in this file: untrusted and trusted. When the trusted applications were split into platform, shared, media, and release applications, the AVRs regarding the use of binder were altered. In that alteration, it became possible for untrusted applications to perform binder IPC to other untrusted applications. In fact any application in the `appdomain` could now perform binder IPC to any other application in the `appdomain`. As noted earlier, every application on the device is a member of the `appdomain`, so every application could now perform binder IPC to every other application.

We reverted this change in the `app.te` file. This involved adding rules allowing the following:

- The `appdomain` can perform binder IPC with each of the trusted application domains.
- Each trusted application domain can perform binder IPC with every other trusted application domain.

- Each trusted application domain can perform binder IPC with the `appdomain`

Thus, the only thing now disallowed is for untrusted applications performing binder IPC with other untrusted applications. Without this change, it is impossible to place restrictions on what applications can use the binder channel. As shown in Figure 26, if this change was not made, then communication between `view0_app` and `view1_app` would be allowed.

```
Information flows into view1_app from view1_app
Objects classes for IN flows:
     fd
     allow appdomain appdomain : fd use ;
```

Figure 26. Apol analysis of information flow using the default `app.te` file

2. Seapp_contexts

The `seapp_contexts` file was updated to allow for the distinct labeling of the separate applications. Four statements were added; one for each of the applications.

```
user=app_* name=com.poc.displayapp domain=display_app
     type=app_data_file levelFromUid=true
user=app_* name=com.poc.trustedcontroller
     domain=controller_app type=app_data_file
     levelFromUid=true
user=app_* name=com.poc.view0 domain=view0_app
     type=app_data_file levelFromUid=true
user=app_* name=com.poc.view1 domain=view1_app
     type=app data file
```

Figure 27. Additions to seapp_contexts

As Figure 27 shows, the name string in each statement refers to one of the application packages. Each application process will be labeled with its own distinct domain. The files are labeled with the same type, but will be separated by the category labeling done by levelFromUid.

A quick check with *adb shell ps –Z* gives us the following security contexts for the processes:

```
u:r:display_app:s0:c37          com.poc.displayapp
u:r:controller_app:s0:c36       com.poc.trustedcontroller
u:r:view0_app:s0:c34            com.poc.view0
u:r:view1_app:s0:c35            com.poc.view1
```

Figure 28. Process security contexts

And similarly, `adb ls –Z /data/data`, for the file contexts:

```
u:object_r:app_data_file:s0:c37  com.poc.displayapp
u:object_r:app_data_file:s0:c36  com.poc.trustedcontroller
u:object_r:app_data_file:s0:c34  com.poc.view0
u:object_r:app_data_file:s0:c35  com.poc.view1
```

Figure 29. File security contexts

3. Poc_app.te

The `poc_app.te` file contains the AVRs for the proof-of-concept applications. There are only a handful of rules needed. Each new application domain must be associated with the base domain as well as the `appdomain`. This is accomplished with the following two statements:

```
type display_app, domain;
app_domain(display_app)
```

Figure 30. `poc_app.te` domain associations

Additionally, a number of binder permissions are used to allow the applications to communicate with the other applications they need to communicate with. The display application must be able to perform binder IPC with the trusted controller. The trusted controller must perform binder IPC with each of the calendar applications. These rules can be accomplished using three binder macros: `binder_use(domain)`, `binder_call(clientdomain, servicedomain)`, and `binder_transfer(clientdomain, servicedomain)`.

4. Mac_permissions.xml

Updating the `mac_permissions.xml` is entirely optional for developers. That being said, it is a good way to improve the security of applications. In our proof-of-concept, the applications use custom permissions to restrict who can talk to whom. That doesn't stop a potentially malicious application from simply reading the `AndroidManifest.xml` files and putting that permission in its own manifest. By updating the `mac_permissions.xml` file, those custom permissions can be further protected by only allowing certain applications to have those permissions.

As discussed in the previous chapter, the `mac_permissions.xml` file can allow for packages to specify their own allow or deny permission rules. The proof-of-concept application packages were added and their corresponding permissions marked allowed. Additionally, under the default scheme, all permissions used by the new packages were denied.

F. SE LINUX POLICY ANALYSIS

It turns out that traditional tools like Apol are not all that useful when analyzing policies for SE Android. This is primarily because of the strong separation that SE Android aims for. There is very little domain transitioning done, and when it does occur, it is usually some system process like `zygote` or `installd` that is doing a single, one way, transition into an application domain. Further, third party applications do not typically communicate with other third party applications so the sharing of files does not occur often. Nevertheless, we can verify that there are no information flows that violate our security goals. We used Apol and Qisaq to evaluate the policy.

1. Apol

We can verify that there is no information flows between the `view0_app` and `view1_app` domains by doing a transitive information flow analysis in Apol. It yields four flows which, when scrutinized, are not of a concern. The first flow is as follows:

```
Flow 1 requires 1 step(s).
view1_app -> view0_app
allow appdomain domain:file {ioctl read getattr lock
     open}; [Disabled]
allow appdomain domain:dir {ioctl read getattr search
     open}; [Disabled]
allow appdomain domain:lnk_file {ioctl read getattr
     lock open}; [Disabled]
```

Figure 31. Information flow from Apol for `view1_app` to `view0_app`

All of those flows are disabled due to a SE Boolean. Unfortunately, Apol does not have the capability to show which Boolean it is without doing some digging. Looking through Apol's policy rules and conditional expressions tab, it turns out that the Boolean governing these flows is `the android_cts` Boolean. This Boolean would only be enabled when the Android Compatibility Test Suite, which is used by developers to test their apps on various devices, is being used, so these AVRs should never be enabled in real-world use. The other three flows involve transitioning through the `mlstrustedsubjects`, `debuggerd` and `zygote`. Being `mlstrustedsubjects`, these are assumed to be trusted so those flows should not be allowed by these subjects.

We can further verify that communication is allowed from the `display_app` domain to the `controller_app` domain, and similarly for the `controller_app` domain to the `view0_app` and `view1_app` domains:

```
allow display_app controller_app : fd use;
 allow controller_app view0_app : fd use;
 allow controller_app view1_app : fd use;
```

Figure 32. AVRs allowing flow from `display_app` to `controller_app` and `controller_app` to `view0_app` and `view1_app`

These allow for the corresponding applications to use file descriptors that are passed via the binder connections. This allows for the passing of data between the applications.

2. Qisaq

Qisaq, as mentioned earlier, is a Python interface to SETools. Qisaq works by importing SE Linux policies and constructing directed graphs with the types being the nodes and the AVRs and domain transitions representing edges. For large policies, like the SE Linux reference policy, the conversion to graphs uses a large amount of both memory and time. Fortunately, the SE Android policy is small enough to be analyzed in a reasonable amount of time.

Qisaq, like Apol, can detect direct information flows. Additionally, it will provide the constraints that restrict flow between two domains. The SE Android policy only contains MLS constraints, and as none of the proof-of-concept applications are mlstrustedsubjects nor do any dominate the others, these constraints do not apply. Direct information flow from `view0_app` to `view1_app` is as follows:

```
Information can flow directly from view0_app to view1_app:
view0_app can invoke class lnk_file read operations
     {getattr, read} on view1_app
view0_app can invoke class dir read operations
     {getattr, read, search} on view1_app
view0_app can invoke class file read operations
     {getattr, read} on view1_app
view0_app can invoke class lnk_file unmapped
     operations {open} on view1_app
```

Figure 33. Qisaq information flow between `view0_app` and `view1_app`

All of these flows are denied by a Boolean as mentioned earlier, but Qisaq does not yet account for them when doing the graph analysis.

Qisaq can also look for indirect influences between two domains. It does so by looking for paths going through types that are not considered trusted mediators. To do this analysis, all `mlstrustedsubjects` were added to the set of trusted mediators. Then, Qisaq does a ‘breakout’ analysis looking for paths between two domains going through types not included in the set. Doing this analysis for `view0_app` and `view1_app` yields no paths outside the mediators. As noted in the Apol section, there

were information flows from `view0_app` to `view1_app` through `debuggerd` and `zygote`. Both of these are `mlstrustedsubjects` and are included in the set of trusted mediators.

G. DISCUSSION

As mentioned earlier, using the current architecture it is possible for data to pass from `view0_app` to the `view1_app` through the allowed communication channels. We propose an alternate architecture to eliminate this.

The proposal is to add a new application to work alongside each of the calendar applications. This new application will access the content provider implemented by the original calendar applications with only read permissions. We now have two applications interacting with the content provider: application A, the original calendar application, and application B, the new application. Application B would now be the one that the trusted controller interacts with when interacting in a restricted mode. In other words, if the mode selected was view0, then the trusted controller would interact with application B of view1. With application B only having read permission to the content provider, it would not be able to update tables with information obtained from view0. This new architecture is represented in Figure 34.

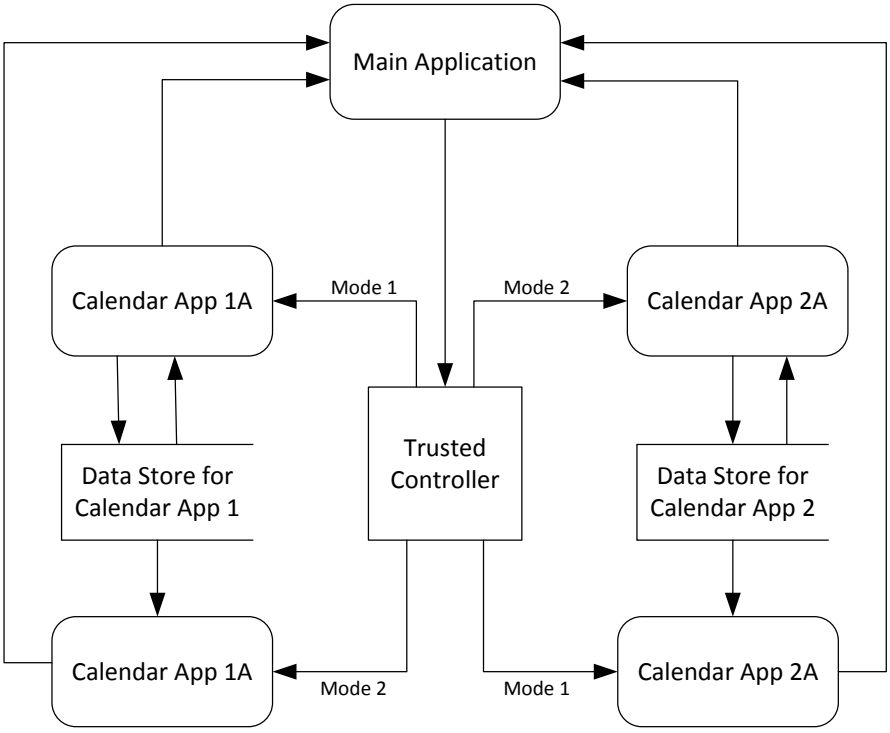

Figure 34. New architecture. Only the two flows of the same mode occur at the same time.

We can ensure that the trusted controller only connects to application A or B appropriately by using a new SE Boolean. By having the main application set the Boolean upon choosing a mode, we can ensure that the trusted controller can never write data to the data stores of view0 and view1 at the same time. The policy for the trusted controller would be adjusted accordingly:

```
bool view0 false;
bool view1 false;
type controller_app, domain;
app_domain(controller_app)
binder_use(controller_app)
if (view0) {
binder_call(controller_app, view0_appA)
binder_transfer(controller_app, view0_appA)
}
if (view1) {
binder_call(controller_app, view1_appA)
binder_transfer(controller_app, view1_appA)
}
binder_call(controller_app, view0_appB)
binder_transfer(controller_app, view0_appB)
binder_call(controller_app, view1_appB)
binder_transfer(controller_app, view1_appB)
```

Figure 35. Proposed policy change for new architecture

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSION

A. FUTURE WORK

This thesis contributes to the currently limited literature on SE Android. With SE Android still being in its infancy, there is much more work that can and should be done. This section mentions some possible future work to be done on SE Android that is not necessarily an extension of this thesis.

The architecture proposed at the end of Chapter VI should be explored further. The reason this architecture was not implemented was due to some challenges in implementing the setting of the SE Boolean. Access to the Java library for `android.os.SELinux` is not readily available for importing into an IDE (Integrated Development Environment). We attempted to include the package needing access to it in the build path and rebuilding the image. However, we were unable to get the Boolean to properly set as needed by the architecture. Further exploration of the SE Manager application source code may reveal why we failed.

The continued development of our proof-of-concept into a full-fledged calendar manager raises many questions. We must consider how the calendars will be updated and ensure that the data remains isolated between the calendars. The architecture proposed above helps to facilitate this. Additionally, one may want to consider how to secure the data in the calendars. Some form of authentication should be used to control access to each of the calendars. One way of doing this is to use a login and password to protect the calendars from unauthorized access. Separate passwords could be used for each of the calendars. In adding this authentication, a display mode could be made available for unauthorized users having limited access to both calendars. Our initial thought is to not allow even this limited access as it provides potential adversaries with some information. However, in discussing some potential features below, that mode may be useful. One may also wish to consider encrypting the calendars' data stored on the phone. This will help protect the data in the event of an adversary having physical access to the phone.

Other calendar features also raise some questions on how to maintain the security goals. For instance, many calendars have an alarm feature. One should consider how this would work in our implementation. The alarm needs to have some access to the data to be able to notify the user. Is it sufficient to have only the limited access mode? In this case it would allow the alarm to function without having the need for the user to be authenticated with one of the calendars. This is an interesting issue that should be explored further.

As mentioned in Chapter VI, our proof-of-concept stores its data locally. In the real world, it is more likely for this data to be stored remotely (in the cloud) so that it can be accessed from multiple devices. Instead of the data coming from content providers managed by the application, the data now comes over a network connection from a storage location not directly managed. This may have an impact on the architecture and whether the security goals can still be reached. How can the proof of concept be changed to maintain its security goals? Is it as simple as replacing the functions in the separate calendar applications that access the content providers with functions that access the cloud storage? That would allow for the general architecture to remain intact, but maybe not the proposed modified architecture. Does the way the calendar is stored in the cloud allow for a read only mode? Should this data be stored locally in the calendars or can it simply be sent to the display and discarded? Many questions arise when considering the use of the cloud.

We mentioned in Chapter VI that we are assuming that the trusted controller behaves properly. This particular element of the proof-of-concept will need to be formally verified and analyzed if it were to be used in the full implementation. Requests to the trusted controller must be handled properly so as to not allow data to flow between the separate calendars applications.

Apart from questions regarding the calendar application, there are questions related to the use of SE Android. As Smalley mentions in [10], SE Android has not been specifically evaluated or approved for use. A considerable amount of work needs to be done on SE Android before this can happen. One example is that an in depth analysis of altered Android code that facilitates SE Android should be performed; particularly for

processes such as `zygote` to verify that the labeling of forked processes is done correctly.

A more formal analysis of the finalized SE Android policy would also need to be done. As of the writing of this thesis the policy is still a work in progress. It may be some time before a finalized version of the policy is available for complete analysis.

B. SUMMARY

Chapter V described the mechanisms that SE Android implements to improve Android security. The features mentioned were the traditional SE Linux MAC, install-time MAC, tag propagation, and permission revocation. It went into further detail on the organization of the SE Android policy and the unique additions that differentiate it from traditional SE Linux.

The proof-of-concept application in Chapter VI demonstrates how to edit the SE Android policy files to enforce security goals for custom applications. Creating policies for new applications is fairly straightforward. Labeling of new application packages can be done in the `seapp_contexts` file. SE Android provides domains and macros to make it easy to obtain the base set of permissions required for applications to operate. MAC can also be applied to the Android permissions; allowing or denying only specified application packages access to certain permissions.

In Chapter VI, we also identify a potential weakness in the default SE Android policy. The default policy allows for any third party application to perform binder calls to any other third party application. We modify the `app.te` file to enable restrictions to be placed on the use of binder channels.

While SE Android is meant to be a security extension transparent to application developers and users, this thesis demonstrates how it can be customized for use on a specific set of applications to improve security.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. PROOF OF CONCEPT CODE

The Java classes and Android manifests for the applications developed for the proof-of-concept application.

A. PACKAGE COM.PROC.DISPLAYAPP

1. MainActivity.java

```
package com.poc.displayapp;

import android.os.Bundle;
import android.app.Activity;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.RadioGroup;
import android.widget.Button;
import android.content.Intent;
import android.util.Log;

public class MainActivity extends Activity {
        private static String TAG = “main.activity”;
        private RadioGroup viewGroup;
        private Button button1;
        private int view = -1;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
         viewGroup = (RadioGroup) findViewById(R.id.radioGroup1);
         button1 = (Button) findViewById(R.id.button1);
      requestView();
   }

   public void requestView() {
         button1.setOnClickListener(new OnClickListener() {
                  @Override
                  public void onClick(View v) {
                           switch (viewGroup.getCheckedRadioButtonId()) {
                                    case R.id.radio0:
                                             view = 0;
                                             break;
                                    case R.id.radio1:
```

```
                                        view = 1;
                                        break;
                        }
                        Intent i = new Intent(getApplicationContext(),
DisplayActivity.class);
                        i.putExtra("view," view);
                        System.out.println("mview: " + view);
                        Log.d(TAG, "Starting display");
                        startActivity(i);
                }
        });
   }
}
```

2. DisplayActivity.java

```
package com.poc.displayapp;

import com.poc.aidl.ItcService;
import java.util.ArrayList;

import android.os.Bundle;
import android.os.RemoteException;
import android.app.Activity;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TableLayout;
import android.widget.TableRow;
import android.widget.TableRow.LayoutParams;
import android.widget.TextView;
import android.content.BroadcastReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.ServiceConnection;
import android.graphics.Color;
import android.os.IBinder;

public class DisplayActivity extends Activity {
        private Button b;
        private BroadcastReceiver receiver;
        private IntentFilter intentFilter;
        ItcService mService;
        boolean mBounded;
```

```
        private int vmode = -1;

        private ServiceConnection mConnection = new ServiceConnection() {

                @Override
        public void onServiceDisconnected(ComponentName name) {
                mBounded = false;
                mService = null;
        }
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
                mService = ItcService.Stub.asInterface(service);
                mBounded = true;
        }
};

        @Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_display);
   b = (Button)findViewById(R.id.button1);

   b.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
                sendRequest(vmode);
      }
   });

   receiver = new BroadcastReceiver() {
      @Override
      public void onReceive(Context context, Intent intent) {
                updateTable(intent);
                }
   };
      }

      @Override
      public void onStart() {
                super.onStart();

      intentFilter = new IntentFilter("com.poc.displayapp.vDisplay");
      this.registerReceiver(this.receiver, intentFilter);

   Intent sIntent = getIntent();
```

```
        vmode = sIntent.getIntExtra("view," -1);

        Intent serviceIntent = new Intent();
        serviceIntent.setClassName("com.poc.trustedcontroller,"
"com.poc.trustedcontroller.MainService");
            boolean ok = bindService(serviceIntent, mConnection,
Context.BIND_AUTO_CREATE);
            Log.v("bound ok? ," String.valueOf(ok));

    }

    @Override
    protected void onResume() {
            super.onResume();
            this.registerReceiver(this.receiver, intentFilter);
    }

    @Override
    public void onStop() {
            super.onStop();
            if (mBounded) {
                    unbindService(mConnection);
                    mBounded = false;
            }
    }

    @Override
    protected void onPause() {
            this.unregisterReceiver(receiver);
            if (mBounded) {
                    unbindService(mConnection);
                    mBounded = false;
            }
            super.onPause();
    }

            private void sendRequest(int view) {
                    Intent request = new Intent("com.poc.trustedcontroller.MainService");
                    request.putExtra("view," view);
                    try {
                            mService.readTables(request);
                    } catch (RemoteException e) {
                            Log.e("RemoteException," e.toString());
                    }
            }
```

```
    public void updateTable(Intent i) {

            int origin = i.getIntExtra("source," -1);
    ArrayList<String> dates = i.getStringArrayListExtra("dates");
    ArrayList<String> appointments = i.getStringArrayListExtra("appointments");
    TableLayout tl = new TableLayout(this);

    if (origin == 0) {
       tl = (TableLayout)findViewById(R.id.tableLayout1);
    }
    else if (origin == 1) {
       tl = (TableLayout)findViewById(R.id.tableLayout2);
    }
    updateRows(tl, dates, appointments);
       }

       @SuppressWarnings("deprecation")
       public void updateRows(TableLayout tlayout, ArrayList<String> dates,
                   ArrayList<String> appointments) {
            System.out.println("size " + dates.size());
         for (int j=0; j < dates.size(); j++) {
            TableRow tr = new TableRow(this);
                TextView date = new TextView(this);
                TextView data = new TextView(this);
            tr.setLayoutParams(new LayoutParams(
                   LayoutParams.FILL_PARENT,
                   LayoutParams.WRAP_CONTENT));
            tr.setBackgroundColor(Color.GRAY);
               tr.setId(100+j);
               date.setId(200+j);
               data.setId(300+j);
            date.setText(dates.get(j));
               data.setText(appointments.get(j));
               tr.addView(date);
               tr.addView(data);
               tlayout.addView(tr, new TableLayout.LayoutParams(
                           LayoutParams.FILL_PARENT,
                   LayoutParams.WRAP_CONTENT));
         }
       }
}
```

3. AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.poc.displayapp"
   android:versionCode="1"
   android:versionName="1.0" >

   <uses-sdk
      android:minSdkVersion="8"
      android:targetSdkVersion="15" />

         <uses-permission android:name="com.poc.trustedcontroller.tcpermission" />
         <uses-permission android:name="com.poc.view0.v0receive" />
         <uses-permission android:name="com.poc.view1.v1receive" />

   <application
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >
      <activity
         android:name=."MainActivity"
         android:label="@string/title_activity_main" >
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>

      <activity
         android:name=."DisplayActivity"
         android:label="@string/title_activity_display" >
         <intent-filter>
            <action android:name="com.poc.displayapp.vDisplay" />
         </intent-filter>
      </activity>
   </application>
</manifest>
```

B. PACKAGE COM.POC.TRUSTEDCONTROLLER

1. MainService.java

```
package com.poc.trustedcontroller;

import com.poc.aidl.ItcService;
import com.poc.aidl.ItcService.Stub;
import com.poc.v0Service.Iv0Service;
import com.poc.v1Service.Iv1Service;
import java.util.ArrayList;
import android.app.Service;
import android.os.IBinder;
import android.os.RemoteException;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.util.Log;

public class MainService extends Service {
	private static String TAG = “tc.service”;
	static final int VIEW0 = 0;
	static final int VIEW1 = 1;
	private static final String action0 = “com.poc.view0.View0Service”;
	private static final String action1 = “com.poc.view1.View1Service”;
	Iv0Service v0Service;
	Iv1Service v1Service;
	boolean mBounded0;
	boolean mBounded1;

	private ServiceConnection mConnection0 = new ServiceConnection() {

		@Override
		public void onServiceDisconnected(ComponentName name) {
			mBounded0 = false;
			v0Service = null;
		}

		@Override
		public void onServiceConnected(ComponentName name, IBinder service)
		{
			v0Service = Iv0Service.Stub.asInterface(service);
			mBounded0 = true;
		}
	};
```

```
	private ServiceConnection mConnection1 = new ServiceConnection() {

		@Override
		public void onServiceDisconnected(ComponentName name) {
			mBounded1 = false;
			v1Service = null;
		}

		@Override
		public void onServiceConnected(ComponentName name, IBinder service)
		{
			v1Service = Iv1Service.Stub.asInterface(service);
			mBounded1 = true;
		}
	};

	@SuppressWarnings("deprecation")
	@Override
	public void onStart(Intent intent, int startId) {
		super.onStart(intent, startId);
	}

	@Override
	public void onCreate() {
		super.onCreate();
		Log.d(TAG, "TCService created");

		Intent serviceIntent1 = new Intent();
		serviceIntent1.setClassName("com.poc.view0,"
"com.poc.view0.View0Service");
		boolean ok1 = bindService(serviceIntent1, mConnection0,
Context.BIND_AUTO_CREATE);
		Log.v("bound ok1? ," String.valueOf(ok1));
		Intent serviceIntent2 = new Intent();

		serviceIntent2.setClassName("com.poc.view1,"
"com.poc.view1.View1Service");
		boolean ok2 = bindService(serviceIntent2, mConnection1,
Context.BIND_AUTO_CREATE);
		Log.v("bound ok2? ," String.valueOf(ok2));

	}

	@Override
```

```
public IBinder onBind(Intent intent) {
        return mBinder;
}

private final ItcService.Stub mBinder = new Stub() {

public void insertTable(Intent intent) {
        int view = intent.getIntExtra(“view,” -1);
        ArrayList<String> columns = intent.getStringArrayListExtra(“columns”);
        ArrayList<String> data = intent.getStringArrayListExtra(“data”);
        Intent i = new Intent();
        if (view == 0) {
                i.setAction(action0);
                i.putExtra(“view,” view);
        } else if (view == 1) {
                i.setAction(action1);
                i.putExtra(“view,” view);
        }
        startService(i);
        }

public void readTables(Intent intent) {
        int view = intent.getIntExtra(“view,” -1);
        Intent i = new Intent(action0);
        i.putExtra(“view,” view);
        Intent j = new Intent(action1);
        j.putExtra(“view,” view);
        try {
                v0Service.queryTables(i);
                v1Service.queryTables(j);
        } catch (RemoteException e) {
                Log.e(“RemoteException,” e.toString());
        }
}
};

@Override
public void onDestroy() {
        if (mBounded0) {
                unbindService(mConnection0);
                mBounded0 = false;
        }
        if (mBounded1) {
                unbindService(mConnection1);
                mBounded1 = false;
```

```
                }
                stopService(new Intent(action0));
                stopService(new Intent(action1));
                super.onDestroy();

        }
}
```

2. TcService.java

```
package com.poc.trustedcontroller;

import android.content.Intent;

public interface TcService {
        void readTables(Intent intent);
        void updateTable(Intent intent);
}
```

3. Android.Manifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.poc.trustedcontroller"
   android:versionCode="1"
   android:versionName="1.0" >
   <uses-sdk
      android:minSdkVersion="8"
      android:targetSdkVersion="15" />
  <permission
     android:name="com.poc.trustedcontroller.tcpermission"
     android:label="tcpermission"
     android:protectionLevel="dangerous" />

  <uses-permission android:name="com.poc.view0.v0permission" />
  <uses-permission android:name="com.poc.view1.v1permission" />

   <application
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name"
      android:theme="@style/AppTheme" >
      <service
         android:permission="com.poc.trustedcontroller.tcpermission"
         android:enabled="true"
         android:name="MainService" >
         <intent-filter>
            <action android:name="com.poc.trustedcontroller.MainService" />
```

```
        </intent-filter>
      </service>
   </application>
</manifest>
```

C. PACKAGE COM.POC.VIEW0

The files for com.poc.view1 are identical to com.poc.view0 sans the name changes, so they will not be shown.

1. View0Service.java

```
package com.poc.view0;

import java.util.ArrayList;
import com.poc.v0Service.Iv0Service;
import com.poc.v0Service.Iv0Service.Stub;
import android.app.Service;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.IBinder;
import com.poc.view0.view0provider;

public class View0Service extends Service {

        @SuppressWarnings(“deprecation”)
        @Override
        public void onStart(Intent intent, int startId) {
                super.onStart(intent, startId);
        }

        @Override
        public void onCreate() {
                super.onCreate();
        }

        @Override
        public IBinder onBind(Intent intent) {
                return mBinder;
        }

        private final Iv0Service.Stub mBinder = new Stub() {

                public void queryTables(Intent intent) {
```

```
int view = intent.getIntExtra("view," -1);
Uri curi = view0provider.CONTENT_URI;
ArrayList<String> dates = new ArrayList<String>();
ArrayList<String> appointments = new ArrayList<String>();

if (view == 0) {
	String[] columns = {
		DatesTable.COLUMN_DATE,
		DatesTable.COLUMN_COMMENT
	};
	Cursor c = getContentResolver().query(curi, columns, null,
null, null);

	if (c.moveToFirst()) {
		do {
			String d =
c.getString(c.getColumnIndex(DatesTable.COLUMN_DATE));
			String a =
c.getString(c.getColumnIndex(DatesTable.COLUMN_COMMENT));
			dates.add(d);
			appointments.add(a);
		} while (c.moveToNext());
	}
	c.close();
} else {
	String[] columns = {
		DatesTable.COLUMN_DATE
	};
	Cursor c = getContentResolver().query(curi, columns, null,
null, null);
	if (c.moveToFirst()) {
		do {
			String d =
c.getString(c.getColumnIndex(DatesTable.COLUMN_DATE));
			String a = "redacted";
			dates.add(d);
			appointments.add(a);
		} while (c.moveToNext());
	}
	c.close();
}
String action = "com.poc.displayapp.vDisplay";
String permission = "com.poc.view0.v0receive";
Intent i = new Intent(action);
i.putExtra("dates," dates);
```

```
                    i.putExtra("appointments," appointments);
                    i.putExtra("source," 0);
                    sendBroadcast(i, permission);
                }
        };
}
```

2. view0provider.java

```
package com.poc.view0;

import java.util.Arrays;
import java.util.HashSet;
import android.content.ContentProvider;
import android.content.ContentResolver;
import android.content.ContentValues;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.text.TextUtils;
import com.poc.view0.DBHelper;

public class view0provider extends ContentProvider {

        private DBHelper database;
        private static final int DATES = 1;
        private static final int DATES_ID = 2;
        private static final String AUTHORITY = "com.poc.view0";
        private static final String BASE_PATH = "dates";
        public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY
+ "/" + BASE_PATH);
        public static final String CONTENT_TYPE =
ContentResolver.CURSOR_DIR_BASE_TYPE + "/dates";
        public static final String CONTENT_ITEM_TYPE =
ContentResolver.CURSOR_ITEM_BASE_TYPE + "/date";

        private static final UriMatcher sURIMatcher = new
UriMatcher(UriMatcher.NO_MATCH);
        static {
                sURIMatcher.addURI(AUTHORITY, BASE_PATH, DATES);
                sURIMatcher.addURI(AUTHORITY, BASE_PATH + "/#," DATES_ID);
        }

        @Override
```

```
	public boolean onCreate() {
		database = new DBHelper(getContext());
		return true;
	}

	@Override
	public Cursor query(Uri uri, String[] projection, String selection,
			String[] selectionArgs, String sortOrder) {
		SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
		checkColumns(projection);
		qb.setTables(DatesTable.TABLE_DATES);
		int uriType = sURIMatcher.match(uri);
		switch (uriType) {
		case DATES:
			break;
		case DATES_ID:
			qb.appendWhere(DatesTable.COLUMN_ID + "=" +
uri.getLastPathSegment());
			break;
		default:
			throw new IllegalArgumentException("Unknown URI: " + uri);
		}
		SQLiteDatabase db = database.getWritableDatabase();
		Cursor c = qb.query(db, projection, selection, selectionArgs, null, null,
sortOrder);
		c.setNotificationUri(getContext().getContentResolver(), uri);
		return c;
	}

	@Override
	public int delete(Uri uri, String arg1, String[] arg2) {
		int uriType = sURIMatcher.match(uri);
		SQLiteDatabase db = database.getWritableDatabase();
		int rowsDeleted = 0;
		switch (uriType) {
		case DATES:
			rowsDeleted = db.delete(DatesTable.TABLE_DATES, arg1, arg2);
			break;
		case DATES_ID:
			String id = uri.getLastPathSegment();
			if (TextUtils.isEmpty(arg1)) {
				rowsDeleted = db.delete(DatesTable.TABLE_DATES,
DatesTable.COLUMN_ID + "=" + id, null);
			} else {
```

```
                        rowsDeleted = db.delete(DatesTable.TABLE_DATES,
DatesTable.COLUMN_ID + "=" + id + " and " + arg1, arg2);
                    }
                    break;
                default:
                    throw new IllegalArgumentException("Unkown URI: " + uri);
                }
                getContext().getContentResolver().notifyChange(uri, null);
                return rowsDeleted;
        }

        @Override
        public String getType(Uri uri) {
                return null;
        }

        @Override
        public Uri insert(Uri uri, ContentValues values) {
                int uriType = sURIMatcher.match(uri);
                SQLiteDatabase db = database.getWritableDatabase();
                long id = 0;
                switch (uriType) {
                case DATES:
                    id = db.insert(DatesTable.TABLE_DATES, null, values);
                    break;
                default:
                    throw new IllegalArgumentException("Unkown URI: " + uri);
                }
                getContext().getContentResolver().notifyChange(uri, null);
                return Uri.parse(BASE_PATH + "/" + id);
        }

        @Override
        public int update(Uri uri, ContentValues values, String selection,
                    String[] selectionArgs) {
                int uriType = sURIMatcher.match(uri);
                SQLiteDatabase db = database.getWritableDatabase();
                int rowsUpdated = 0;
                switch (uriType) {
                case DATES:
                    rowsUpdated = db.update(DatesTable.TABLE_DATES, values,
selection, selectionArgs);
                    break;
                case DATES_ID:
                    String id = uri.getLastPathSegment();
```

```
                        if (TextUtils.isEmpty(selection)) {
                                rowsUpdated = db.update(DatesTable.TABLE_DATES,
values, DatesTable.COLUMN_ID + "=" + id, null);
                        } else {
                                rowsUpdated = db.update(DatesTable.TABLE_DATES,
values, DatesTable.COLUMN_ID + "=" + id + " and " + selection, selectionArgs);
                        }
                        break;
                default:
                        throw new IllegalArgumentException("Unknown URI: " + uri);
                }
                getContext().getContentResolver().notifyChange(uri, null);
                return rowsUpdated;
        }

        private void checkColumns(String[] projection) {
                String[] available = { DatesTable.COLUMN_ID,
DatesTable.COLUMN_DATE, DatesTable.COLUMN_COMMENT };
                if (projection != null) {
                        HashSet<String> requestedColumns = new
HashSet<String>(Arrays.asList(projection));
                        HashSet<String> availableColumns = new
HashSet<String>(Arrays.asList(available));
                        if (!availableColumns.containsAll(requestedColumns)) {
                                throw new IllegalArgumentException("Unknown columns
in projection");
                        }
                }
        }
}
```

3. DBHelper.java

```
package com.poc.view0;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;

public class DBHelper extends SQLiteOpenHelper {

        private static final String DATABASE_NAME = "dates.db";
        private static final int DATABASE_VERSION = 1;


        public DBHelper(Context context) {
```

```
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase database) {
        DatesTable.onCreate(database);
    }

    @Override
    public void onUpgrade(SQLiteDatabase database, int oldVersion, int
newVersion) {
        DatesTable.onUpgrade(database, oldVersion, newVersion);
    }

}
```

4. DatesTable.java

```
package com.poc.view0;

import android.database.sqlite.SQLiteDatabase;
import android.util.Log;

public class DatesTable {

    public static final String TABLE_DATES = “dates”;
    public static final String COLUMN_ID = “_id”;
    public static final String COLUMN_DATE = “date”;
    public static final String COLUMN_COMMENT = “comment”;
    private static final String DATABASE_CREATE = “create table “ +
TABLE_DATES + “(“ +
                    COLUMN_ID + “ integer primary key autoincrement, “ +
                    COLUMN_DATE + “ text not null, “ +
                    COLUMN_COMMENT + “ text);”;

    private static final String INSERT1 = “insert into “ + TABLE_DATES +
                    “ (“ + COLUMN_DATE + ,” “ + COLUMN_COMMENT + “)” +
                    “ values (‘Sept1’, ‘Blah’);”;
    private static final String INSERT2 = “insert into “ + TABLE_DATES +
                    “ (“ + COLUMN_DATE + ,” “ + COLUMN_COMMENT + “)” +
                    “ values (‘Sept2’, ‘Bleh’);”;
    private static final String INSERT3 = “insert into “ + TABLE_DATES +
                    “ (“ + COLUMN_DATE + ,” “ + COLUMN_COMMENT + “)” +
                    “ values (‘Sept3’, ‘Bluh’);”;

    public static void onCreate(SQLiteDatabase db) {
```

```
                db.execSQL(DATABASE_CREATE);
                db.execSQL(INSERT1);
                db.execSQL(INSERT2);
                db.execSQL(INSERT3);
        }

        public static void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{
                Log.w(DBHelper.class.getName(), “Upgrading database from version “ +
oldVersion + “ to “ +
                                newVersion + ,” which will destroy all old data”);
                db.execSQL(“drop table if exists “ + TABLE_DATES);
                onCreate(db);
        }

}
```

5. AndroidManifest.xml

```
<manifest xmlns:android=“http://schemas.android.com/apk/res/android”
   package=“com.poc.view0”
   android:versionCode=“1”
   android:versionName=“1.0” >

   <uses-sdk
      android:minSdkVersion=“8”
      android:targetSdkVersion=“15” />
         <permission
            android:name=“com.poc.view0.v0permission”
            android:label=“v0permission”
            android:protectionLevel=“normal” />
         <permission
            android:name=“com.poc.view0.v0receive”
            android:label=“v0receive”
            android:protectionLevel=“normal” />

   <application
      android:icon=“@drawable/ic_launcher”
      android:label=“@string/app_name”
      android:theme=“@style/AppTheme” >
      <service
         android:permission=“com.poc.view0.v0permission”
         android:enabled=“true”
         android:name=.”View0Service” >
         <intent-filter>
            <action android:name=“com.poc.view0.View0Service” />
```

```
            </intent-filter>
        </service>
        <provider
            android:name=.”view0provider”
            android:authorities=“com.poc.view0” >
        </provider>
    </application>
</manifest>
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. SE POLICY

The added or modified policy files used for in conjunction with the proof-of-concept applications:

A. SEAPP_CONTEXTS

```
isSystemServer=true domain=system
user=system domain=system_app type=system_data_file
user=nfc domain=nfc type=nfc_data_file
user=radio domain=radio type=radio_data_file
user=app_* domain=untrusted_app type=app_data_file levelFromUid=true
user=app_* seinfo=platform domain=platform_app type=platform_app_data_file
user=app_* seinfo=shared domain=shared_app type=platform_app_data_file
user=app_* seinfo=media domain=media_app type=platform_app_data_file
user=app_* seinfo=release domain=release_app type=platform_app_data_file
user=app_* seinfo=release name=com.android.browser domain=browser_app
        type=platform_app_data_file
user=app_* name=com.poc.displayapp domain=display_app type=app_data_file
        levelFromUid=true
user=app_* name=com.poc.trustedcontroller domain=controller_app type=app_data_file
        levelFromUid=true
user=app_* name=com.poc.view0 domain=view0_app type=app_data_file
        levelFromUid=true
```

B. POC_APP.TE

```
#
# com.poc.displayapp
#
type display_app, domain;
app_domain(display_app)
binder_use(display_app)
binder_call(display_app, controller_app)
binder_transfer(display_app, controller_app)

#
# com.poc.trustedcontroller
#
type controller_app, domain;
app_domain(controller_app)
binder_use(controller_app)
binder_call(controller_app, view0_app)
binder_transfer(controller_app, view0_app)
```

```
binder_call(controller_app, view1_app)
binder_transfer(controller_app, view1_app)

#
# com.poc.view0
#
type view0_app, domain;
app_domain(view0_app)

#
# com.poc.view1
#
type view1_app, domain;
app_domain(view1_app)
```

C. MODIFICATION TO TE_MACROS

```
+define(`binder_allows', `
+binder_call($1, $2)
+binder_transfer($1, $2)
+binder_call($2, $1)
+binder_transfer($2, $1)
+')
```

D. MODIFICATIONS TO APP.TE

```
-binder_call(appdomain, appdomain)
-binder_transfer(appdomain, appdomain)

+binder_call(appdomain, platform_app)
+binder_transfer(appdomain, platfrom_app)
+binder_call(appdomain, media_app)
+binder_transfer(appdomain, media_app)
+binder_call(appdomain, shared_app)
+binder_transfer(appdomain, shared_app)
+binder_call(appdomain, release_app)
+binder_transfer(appdomain, release_app)
+binder_call(appdomain, shared_app)
+binder_transfer(appdomain, shared_app)
+binder_allows(platform_app, media_app)
+binder_allows(platform_app, shared_app)
+binder_allows(platform_app, release_app)
+binder_allows(media_app, shared_app)
+binder_allows(media_app, release_app)
+binder_allows(shared_app, release_app)
```

E. MODIFICATIONS TO MAC_PERMISSIONS.XML

+<package name="com.poc.displayapp" >
+ <allow-permission name="com.poc.trustedcontroller.tcpermission" />
+ <allow-permission name="com.poc.view0.v0receive" />
+ <allow-permission name="com.poc.view1.v1receive" />
+</package>

+<package name="com.poc.trustedcontroller" >
+ <allow-permission name="com.poc.view0.v0permission" />
+ <allow-permission name="com.poc.view1.v1permission" />
+</package>

Under the <default> tag:
+ <deny-permission name="com.poc.trustedcontroller.tcpermission" />
+ <deny-permission name="com.poc.view0.v0permission" />
+ <deny-permission name="com.poc.view1.v1permission" />
+ <deny-permission name="com.poc.view0.v0receive" />
+ <deny-permission name="com.poc.view1.v1receive" />

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] A. Smith, “46% of American adults are smartphone owners,” Pew Research Center, 1 March 2012.

[2] “May 2012 U.S. Mobile Subscriber Market Share,” comScore, 2 July, 2012.

[3] “The true face of the Android threat,” *Trend Micro*, [online], Available: http://www.trendmicro.co.uk/newsroom/pr/the-true-face-of-the-android-threat/. (Accessed: 20 July 2012).

[4] J. P. Anderson, “Computer Security Technology Planning Study,” Deputy for Command and Management Systems HQ Electronic Systems Division, L.G. Hanscom Field, Bedford, MA, October 1972.

[5] D. F. Ferraiolo and D. R. Kuhn, “Role-Based Access Control,” *Proc. 15th National Computer Security Conf.*, Gaithersburg, MD, 1992, pp. 554–563.

[6] D. F. Ferraiolo, et al., *Role-Based Access Control*, Norwood, MA: Artech House, 2003.

[7] R. S. Sandhu, “Rationale for the RBAC96 family of access control models,” *Proc. 1st ACM Workshop on Role-based access control*, Gaithersburg, MD, 1995.

[8] Handbook of Information Security Management, *cccure.org*, [online], Available: http://www.cccure.org/Documents/HISM/ewtoc.html. (Accessed: 20 August 2012).

[9] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Bedford, MA, ESD-TR-73–278, Vol. III, April 1974.

[10] R. Spencer, et al., “The Flask Security Architecture: System Support for Diverse Security Policies,” *Proc. 8th USENIX Security Symp.*, Washington, D.C., August 23–26, 1999.

[11] R. Whitwam, “Circumventing Google’s Bouncer, Android’s anti-malware system,” *extremetech*, [online], Available: http://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system. (Accessed: 10 June 2012).

[12] SEAndroid, *SELinux Wiki*, [online] 2012, Available: http://selinuxproject.org/page/SEAndroid. (Accessed: 1 September 2012).

[13] S. Smalley. (2011, September). *The Case for SE Android* [Online]. Available: http://selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf.

[14] A. Shabtai, et al. (2010 May–June). "Securing Android-Powered Mobile Devices Using SE Linux," *Security & Privacy, IEEE*, vol. 8, no. 3, pp.36–44.

[15] S. Bugiel, et al. "Practical and Lightweight Domain Isolation on Android," *Proc. 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. Chicago, IL, 2011, pp. 51–62.

[16] W. Enck, et al., "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *Proc. 9th USENIX Security Symp.*, 2011.

[17] S. Bugiel, et al., "A new Android evolution to mitigate privilege escalation attacks," Technische Universitat Darmstadt, Germany, TR-2011–04, 2011.

[18] Android Developers Guide, *developer.android.com*, [online], Available: http://developer.android.com. (Accessed: 20 August 2012).

[19] Android Booting, *elinux.org*, [online], Available: http://elinux.org/

Android_Booting. (Accessed: 1 September 2012).

[20] Google Android SDK, *dmxzone*, [online], Available: http://dmxzone.com/

go/14339/google-android-sdk-released/. (Accessed: 20 August 2012).

[21] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))*, 1003.1–2008.

[22] S. Ante, "Banks rush to fix security flaws in wireless apps," *Wall Street Journal Online*, [online], Available: http://online.wsj.com/article/

[23] Android Riskware, *f-secure*, [online], Available: http://www.f-secure.com/weblog/archives/archive-092011.html. (Accessed: 20 August 2012).

[24] A. Felt, et al. "Android Permissions Demystified," *Proc. 18th ACM Conf. on Computer and Communications Security*, Chicago, IL, 2011, pp. 627–638.

[25] SQLite, *sqlite.org*, [online], Available: http://www.sqlite.org. (Accessed: 1 September 2012).

[26] C. Marforio, et al. "Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems," ETH Zurich, System Security Group, TR-724, 2011.

[27] M. Miller, et al. "Capability Myths Demolished," John Hopkins University, Systems Research Laboratory, Department of Computer Science, 2003.

[28] B. McCarty, *SELinux: NSA's Open Source Security Enhanced Linux*, Sebastopol, CA: O'Reilly Media, Inc., 2005.

[29] D. E. Denning, “A Lattice Model of Secure Information Flow,” *Communications of the ACM*, vol. 19, pp. 236–243, May 1976.

[30] S. Marouf and M. Shehab. “SEGrapher: Visualization-based SE Linux Policy Analysis,” *4th Symposium on Configuration Analytics and Automation*, Arlington, VA, 2011, pp. 1–8.

[31] W. Xu, et al. “Visualization Based Policy Analysis: Case Study in SE Linux,” *Proc. 13th ACM Symp. on Access Control Models and Technologies*, Estes Park, CO, 2008, pp. 165–174

[32] SETools - Policy Analysis Tools for SELinux, *oss.tresys.com*, [online], Available: http://oss.tresys.com/projects/setools. (Accessed: 20 August 2012).

[33] External/sepolicy, *bitbucket*, [online], Available: https://bitbucket.org/seandroid/external-Sepolicy/src. (Accessed: 20 August 2012).

[34] Selinux, *Mailing list archives*, [online], Available: http://marc.info/?l=selinux. (Accessed: 1 September 2012).

[35] J. Oberheide, “Don’t root robots,” *UofM SUMIT_11*, Ann Arbor, MI, 2011.

[36] Setuid(2), Linux Programmer’s Manual, [online], Available: http://www.kernel.org/doc/man-pages/online/pages/man2/setuid.2.html. (Accessed: 20 August 2012).

[37] Getrlimit(2), Linux Programmer’s Manual, [online], Available: http://www.kernel.org/doc/man-pages/online/pages/man2/getrlimit.2.html. (Accessed: 20 August 2012).

[38] CVE-2009–1185, National Vulnerability Database, [online], Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009–1185. (Accessed: 20 August 2012).

[39] Seandroid, *bitbucket*, [online], Available: https://bitbucket.org/seandroid/. (Accessed: June 2012).

[40] S. Bugiel, et al., “Towards Taming Privilege-Escalation Attacks on Android,” *Proc. 19th Annual Network & Distributed System Security Symp.*, 2012.

[41] J. Burns,. “Mobile Application Security on Android, Context on Android Security.” Black Hat, June 30 2009.

[42] E. Chin, et al. “Analyzing Inter-Application Communication in Android,” *Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services*, Bethesda, MD, 2011, pp. 239–252.

[43] L. Davi, et al. “Privilege Escalation Attacks on Android,” *Proc. 13th Int. Conf. on Information Security*, Boca Raton, FL, 2010, pp. 346–360.

[44] W. Enck, et al. (2009 January) “Understanding Android Security,” *Security & Privacy IEEE*, vol. 7, pp. 50–57.

[45] “First Steps with Security-Enhanced Linux (SE Linux): Hardening the Apache Web Server,” IBM Corporation. 2009.

[46] A. Herzog and J. Guttman, “Achieving Security Goals with Security-Enhanced Linux.” MITRE Corporation. February 5, 2002.

[47] L. F. Marin, “SE Linux Policy Management Framework for HIS,” M.S thesis, Queensland University of Technology: QUT Digital Repository, 2008.

[48] M. Nauman, et al. “Apex: Extending Android permission model and enforcement with user-defined runtime constraints,” *Proc. 5th ACM Symp on Information, Computer and Communications Security*, Beijing, China, 2010, pp. 328–332.

[49] C. Orthacker, et al. “Android Security Permissions – Can we trust them?” University of Technology Graz, Institute for Applied Information Processing and Communications, Graz, Austria.

[50] T. Rosa, “Android Binder Security Note: On Passing Binder Through Another Binder,” unpublished.

[51] T. Schreiber, “Android Binder: Android Interprocess Communication,” Seminar thesis, Ruhr-Universität Bochum, October 5, 2011.

[52] S. Smalley, “Configuring the SE Linux Policy,” NSA, January 2003.

[53] X. Zhang, et al., “SEIP: simple and efficient integrity protection for open mobile platforms,” *12th International Conf. on Information and Communications Security*, Barcelona, Spain, 2010.

[54] L. Badger, et al., “Practical Domain and Type Enforcement for UNIX,” *Proc.of the 1995 IEEE Symp. On Security and Privacy*, Washington, D.C., pp. 66.

[55] Fedora, *fedoraproject.org*, [online], Available: http://fedoraproject.org/. (Accessed: September 2012).

[56] Ubuntu, *ubuntu.com*, [online], Available: http://www.ubuntu.com/. (Accessed: September 2012).

[57] Red Hat, *redhat.com*, [online], Available: http://www.redhat.com/. (Accessed: September 2012).

[58] Android, *android.com*, [online], Available: http://www.android.com/. (Accessed: September 2012).

[59] "Copyright Law of the United States and Related Laws Contained in Title 17 of the United States Code," Title 17 *U.S. Code*, Sec. 1201. 2011 ed., 250–259. Print.

[60] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," *Proc. Of the Network and Distributed System Security Symposium*, San Diego, CA, 2005.

[61] GNU M4, gnu.org, [online], Available: http://www.gnu.org/software/m4/. (Accessed: October 2012).

[62] Open Binder, *angryredplanet.com*, [online], Available: http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html/. (Accessed: October 2012).

[63] N. J. Percoco and S. Schulte, "Adventures in BouncerLand," *Black Hat USA 2012*, [online], Available: http://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_U.S._12_Percoco_Adventures_in_Bouncerland_WP.pdf. (Accessed:

November 2012).

[64] Chromium OS, *chromium.org*, [online], Available: http://www.chromium.org/chromium-os (Accessed: November 2012).

[65] EnGarde Secure Linux, *engardelinux.org*, [online], Available: http://www.engardelinux.org/ (Accessed: October 2012).

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. George Dinolt
   Naval Postgraduate School
   Monterey, California

4. Karen Burke
   Naval Postgraduate School
   Monterey, California

5. Stephen Smalley
   National Security Agency
   Ft. Meade, Maryland

6. Matt Benke
   NSA, I4221
   Ft. Meade, Maryland

7. John Loucaides
   NSA, I4221
   Ft. Meade, Maryland

8. John Mildner
   SPAWAR Atlantic
   Charleston, South Carolina

9. Jennifer Guild
   SPAWAR Atlantic
   Charleston, South Carolina